A preprocessor building system for the C language

Hashim, Khairuddin

Award date:
1989

Awarding institution:
University of Bath

A PREPROCESSOR BUILDING SYSTEM

FOR

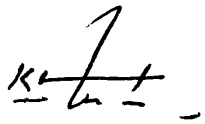
THE C LANGUAGE

Submitted by Khairuddin HASHIM

for the degree of PhD of the University of Bath,

1989.

COPYRIGHT



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

ABSTRACT

This thesis presents an approach to the building of preprocessors for the C Language through a syntax based C Preprocessor Building System (CPBS). It also introduces a Preprocessor Specification Language (PSL) for use in specifying the preprocessors. CPBS accepts a specification of the intended preprocessor and generates the required preprocessor. An interface is also available to help the user specify a preprocessor. A unique method of conveying preprocessor instructions is introduced : using the system, preprocessing instructions are conveyed through standard C language comments. With this approach, preprocessing of other preprocessors can co-exist in an input program with preprocessing instructions of an intended preprocessor but only preprocessing instructions of the intended preprocessor is processed. This means it is possible to pipe together several generated preprocessors to do several different preprocessing tasks consecutively.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor, Dr. Peter Wallis, for training and supervising me during the course of the research work.

A thankful mention also to the staff of the Bath University Computing Unit for all the services rendered, especially to the ever so cheerful Mr. John Gardiner.

I would also like to thank the University of Malaya, Kuala Lumpur and the Public Services Department of Malaysia for giving me the opportunity to further my studies. I hope to give my very best when I return to Malaysia.

Last but not least, I would like to thank my ever so loving and supporting wife, Zurinah, and my children, Naslia and Zuhair, for the sacrifices they had to make to allow me to pursue my quest for knowledge. I thank God for having them.

TABLE OF CONTENTS

CHAPTER ONE

INTRODUCTION

The subject matter of this thesis is preprocessor building through the use of an experimental C Preprocessor Building System (CPBS) for the C language. This thesis describes and demonstrates the system which can build a preprocessor for the user through a specification of the intended preprocessor in a specification file using the Preprocessor Specification Language (PSL). A C language preprocessor that recognises the grammar of C and provides extensions and checks to the language can be built using CPBS with a minimal amount of effort.

This approach to the building of preprocessors opens up a whole new dimension in the building of preprocessors for the purpose of language extension. With the simplicity of specifying macros and new keywords into C, the system provides the easiest possible way of implementing an extension of C. The system provides facilities that are present in text and syntax macro processors.

One of its features is in providing the facility to build preprocessors that recognise special comments containing macros or new keywords. The system also allows the specification of new keywords. It is also

equipped with a run-time link facility that allows for run-time checknig routines to be linked to the output of the preprocessors. A facility for prefixing is also available to allow the user to build preprocessors that have to handle name clashes as a result of the insertions of run-time codes involving variables and functions.

Chapter 2 discusses the types of preprocessors available in the field of language extension and the tools that are available with some emphasis on existing C preprocessor based systems. Chapter 3 describes in detail the PSL specification language and the CPBS system. Chapter 4 demonstrates the capabilities of CPBS as a tool for language extensions. Chapter 5 gives a final example, the introduction of generic packages in C and Chapter 6 discusses the possible areas of research that will be pursued, along the lines of the principles and basis described in this thesis. One of these is a modification of CPBS to provide a general language preprocessor builder for a language, given its grammar productions and lexical entities.

The modifications and extensions on the system which I have forwarded in the final chapter will definitely be pursued in the not too distant future. Finally, I would like to say that I am glad that I have been involved in doing the research. It is my hope that this thesis will be a stepping stone for me to pursue better and more exciting research work in the future.

CHAPTER TWO

PREPROCESSORS AND THEIR TOOLS

This chapter gives a brief discussion of the types of preprocessors currently available and the tools that can be used to build them. It also discusses some of the areas where preprocessors have been used for the purpose of language extensions.

2.1 : Preprocessors

A preprocessor is a program that performs modifications to input data in order to make it suitable for input to another program, typically a compiler. The modifications may be simple changes of format, or may include macro expansions.

Basically, preprocessors can be divided into two types, text preprocessors which handle text as strings of characters with macro facilities for string substitution and expansion; and syntax preprocessors which provides macro facilities based on the syntax of the input. Text preprocessors are normally represented by macro processors while syntax preprocessors are represented by syntax macro processors. Macro

processors are not suitable for use in language extensions as they do not fully validate macro calls and hence can produce replacement texts that contain syntax errors. Some syntax macro processors handle macro substitution and expansion until the syntax analysis stage but most are embedded in the syntax analysis stage of a compiler. Some syntax macro processors are general purpose (language independent) and some are special purpose (language dependent).

2.2 : Preprocessors and Language Extension

As discussed in [CHEA69], there now exist diverse programming language requirements which are becoming continually more diverse; therefore it is of great importance that each user in the spectrum of users be supplied with a language facility appropriate to his problem area. As a language may not contain all the features a user wants, a user might have to change his programming language every time he requires a different type of programming facility.

There have been three rather different approaches used to provide a wide range of language features and facilities. The first one used historically was to provide a large variety of different programming languages to cater for the spectrum of programming needs. It is not only an undoubtedly expensive approach but also does not provide support for hybrid type problems. The second approach is to have a single language that provides all the features and facilities that might be reasonably

required for any problem. This can be said of the language Ada but the sheer size and thus overhead associated with specifying the language, learning the language, compiling, and so on is very high. Also, there will come a time when new features and facilities are to be introduced and modifications to such a language will be difficult to perform. A third approach is to build extensible languages. This was first thought to be the complete salvation for the programmer. However, it was soon discovered that it is not to be so; the main reasons for the failure of extensible languages are as follows. Extensible languages are complex. [BROW79] highlights the problems of having to learn new macro defining languages. It proposes the minimisation of the effort needed to learning macro rather than providing ever more powerful facilities. Similar arguments apply equally to the learning of new extensible languages. Furthermore, the new extensible languages have to compete with well established and heavily invested languages.

How then can we make use of existing languages and provide extensions to them without having to hop from one programming language to another or having to learn a new extensible programming language with complex macro definitions? The answer is partly available through the use of preprocessors for language extensions. The approach taken here is quite similar to the idea of extensible languages except that instead of a language that provides facilities for extensions, we have a preprocessor that extends the language through preprocessing. It will then be quite practical for a user to stick to one language and extend it as

required.

The idea of improved macro processors was first introduced by Cheatham [CHEA66] and Leavenworth [LEAV66]. They brought about the idea of syntax macros. A syntax macro takes advantage of the syntactic structures of programming languages. It would also check that the parameter of a macro satisfy the syntactic class (i.e. statement, identifier, literal etc.) that the macro was defined for. This moves some of the responsibility of syntax analysis from the macro writer to the macro processor.

A number of generalised schemes using this approach were developed, such as MACRO [GREE79] and STEP. Features included in these schemes are extremely flexible macro call formats that allow virtually any form of construct to be specified. [LAYZ85] states that it was becoming increasingly apparent that whilst the generalised syntax macro processors are unlikely to fail on the basis of a poor macro definition language, they may well fail on an inability to deal conveniently with both the fundamental structures of any language and also its peculiarities. As a result, a number of single-language oriented schemes were first introduced. Single-language preprocessors are thus preferred when extensions that blend well with the syntax of a language is required.

We now discuss some C preprocessor based systems that exist. One such system is Modular C [BOYD83]. This is an approach to modularisation in C which does not involve an extension to the existing C language. The technique requires no compiler modification. The C preprocessor plays an important role in this approach. Since the preprocessor scans of the code is part of a standard C compiler's first pass, its ability to do string matching, file scanning, conditional testing and macro definitions are used effectively. In conjunction with C's feature of pointers to functions, this method of modularisation in C creates a clear distinction between structures that form the interfaces between modules and those that represent data manipulations. [DUTT85] presents a simpler implementation of modular programming in C using the standard C preprocessor.

The language C++ [STRO82] is another extension of C by Bjane Stroustrup. It is an enhancement to the C language which supports Simula-style classes and their inheritance properties. Initially, it was implemented as a class preprocessor, an extra pass over the C source text, after the standard C preprocessor. Now, it is being implemented through a new first pass of the C compiler, which replaces the extra pass over the source text with a 'class preprocessor' pass over an internal representation.

In practice, each class is represented by two files. One contains declarations that must be included in every file that uses the class. The other contains definitions, and is compiled, archived and linked during execution by the linker. The entire contents of the first file is therefore public information, while the second is truly private only when compiled into object codes. This makes the public/private distinctions provided by a class declaration something of a misnomer as the definition file can be read from outside the language; even copied and modified to make private information accessible.

Objective C [COX86] is another extension of C implemented through a pass in the C compiler. It is a hybrid language that contains all of C language plus major parts of Smalltalk-80. It supports Object Oriented Programming in that it facilitates the writing of programs that involve objects and messages. However, a theoretical weakness in this hybrid of C is that it is always possible to bypass the object-oriented machinery to access an object's private information directly, which is an infringement to Objective C's basic idea of encapsulation and information hiding.

[GRUN86] presents an idea for introducing generic packages into C. It tries to emulate the structuring achieved by generic packages in Ada but does not do it quite so successfully. The generic specification section is enhanced with the use of comments to make it look similar to a generic specification in Ada. The structuring used solely with the help of the C preprocessor is not sufficient as it is prone to name clashes and only one

instantiation of a generic package is possible. No reference is made to a particular package when a member of it is being referred as there is no actual link between a member of a package and its package. A proper and complete way of introducing generic packages into C is demonstrated in Chapter 5 using CPBS.

2.3. TOOLS

In this section, we discuss some of the tools that are available for use in building preprocessors under the Unix system. The tools to be described here are the standard C Preprocessor (CPP) and the M4 macro processor; and two language development tools, Lex and Yacc.

2.3.1. LEX (Lexical Analyser Generator).

Lex is a software tool in the UNIX environment that lexically processes character input streams. Lex is a program generator that produces a program in a general purpose language that recognises regular expressions. It is specified using a high-level expression language.

Lex :

i) provides a means of recognising regular expressions in the input by

character string matching;

ii) allows the user to provide routines to be executed once a regular expression is recognised;

iii) sends strings that do not match to the output unaltered;

iv) does not provide any mechanism for representing self-referential structures; and

v) generates a function yylex that does the lexical processing.

As it is, Lex can only be used to match expressions described quite explicitly and not anything that is recursive. It is probably well suited for functions of a lexical nature, e.g. string replacement, token recognition, formatting text, etc. and for segmenting input in preparation for a parsing routine.

2.3.2. YACC (Yet Another Compiler-Compiler).

The Yacc program provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process. This includes rules describing the input structure, code to be invoked when these rules are recognised. and a low-level

routine to do basic input. The Yacc program then generates a function to control the input process. This function, called a parser, calls the lexical analyser to pick up the basic items (tokens) from the input stream. These tokens are organised according to the input structure rules, called grammar rules. When one of these rules has been recognised, the user code (action) is invoked. Actions have the ability to return values and make use of values of other actions.

The heart of the input specification is a collection of grammar rules. Each rule describes and names an allowable structure. An important part of the input process is carried out by the lexical analyser which finally passes the tokens to the parser. The lexical analyser recognises terminal symbols while the parser recognises non-terminal symbols.

Yacc by itself represents a parser generator but other structures will be required to use it effectively.

Yacc :

i) imposes a structural hierarchy on its input;

ii) allows the inclusion of user own routines in the form of actions when rules are recognised;

iii) generates a function called a parser that controls the parsing process;

iv) allows structures/rules that are self-referential e.g. the productions of a language; and

v) allows reference to structures that make up the parser stack.

2.3.3 : The C Preprocessor (CPP)

The CPP is basically a text macro processor. It provides facilities for conditional compilation, retrieves files from the file system and; substitutes and expands macros. It is an important tool in its own right and can be combined with other language processors. As the CPP is a macro processor, its capabilities are limited to file and string manipulations. It is therefore not capable of functioning properly where a syntax related function is required.

2.3.4 : The M4 Macro Processor

There is also an M4 macro processor available under the Unix system that provides the following features : i) replacement of one string of text by another; ii) macros that accepts arguments; iii) arithmetic capabilities; iv) file manipulation; v) conditional macro expansion; and vi) string and substring functions. It is quite similar to the CPP with some added facility such as the arithmetic capability and string handling functions.

CHAPTER THREE

THE PREPROCESSOR BUILDING SYSTEM

3.1 : An Overview

This chapter describes in detail the preprocessor building system. It describes the various processes that make up the system. It also discusses the design issues and problems faced when designing and implementing the system. An interface has also been implemented for use in specifying the specification of the intended preprocessor.

The system was built to simplify the building of preprocessors for the C language. Using this system, a user can build a C language preprocessor with a minimal amount of effort. Otherwise, it is a substantial task for a programmer to build a preprocessor that can recognise the grammar of C and provide additional structures or checks for the language. It is important to note that supplied with the grammar productions of a programming language and with some modifications done to the basic lexical and syntax analysis files, preprocessors of that language can then be built. Details of such modifications will be discussed further in Chapter 6.

The system provides facilities to define syntax macro through a simple specification mechanism. It can also be used to provide new keywords that instruct a preprocessor to do certain checks or to represent constructs. It is capable of building preprocessors that do run-time checks on the structures of C. This is done by introducing C checking codes into the input of the preprocessor to be executed during run-time.

The system can also be used to implement syntax-directed translation. It is therefore possible, using the system, to have an implementation of a preprocessor extension of C to process input with additional extension structures and new keywords and produce an output in standard C representing the extension.

This preprocessor building system accepts a specification of the intended preprocessor and generates the required preprocessor. The instructions to the system concerning preprocessing function requirements are given in a specification file known as the user specification file. This specification file is written in a specification language to be discussed in the next section.

Generally, a C preprocessor processes a program input embellished with preprocessing instructions and constructs by inserting code to perform the necessary functions, for e.g. doing static checks, inserting C codes to implement dynamic checks or replacing preprocessing constructs with C constructs. Various different functional tasks are better

implemented using several preprocessors as it is much easier to write relatively simple additional preprocessors rather than complicating an existing one. With the piping facility provided by the Unix system , generated preprocessors can be piped together to do several different preprocessing functions consecutively.

This system makes use of the Unix tools Lex and Yacc in building the preprocessors. They help in simplifying the automatic generation of the preprocessors. The language Awk is also used in processing the specification files.

3.2 : Design and Implementation

3.2.1 : Design Requirements

To provide users with a preprocessor building system, the system should satisfy the following essential requirements :

i) introduce preprocessing instructions;

ii) allow preprocessing instructions for a preprocessor to pass through another preprocessor;

iii) perform preprocessing functions when preprocessing keywords representing instructions are recognised;

iv) insert/substitute/expand text of the input program; and

v) link run-time codes for run-time checking.

Using the system, any reference to the grammar of C should be made with reference to the CPBS C Language Productions of the system which are listed in Appendix II.

Let us consider briefly how the requirements can be implemented. We now consider the first requirement in the list which is the ability to introduce preprocessing instructions. This is done by declaring all preprocessing instructions as new keywords to C through Lex and Yacc. The keywords are then introduced into the grammar of C through specifications of additional productions or extensions of existing ones. Care must be taken to avoid any grammatical ambiguity. Discussions of grammar ambiguities will be referred to later in the chapter.

An essential requirement for the system is transparency between preprocessors. A preprocessor should allow preprocessing instructions of another preprocessor to pass through without any difficulty. This is the next requirement in the list. It can be achieved through the use of the standard C comment.

Comments are handled by the standard C preprocessor in compilers. The use of only Lex and Yacc on the grammar of C normally does not allow comments to pass through. The approach taken is to include preprocessing comments into the grammar of C by modifying the standard C grammar. (Refer to productions in Appendix II). This is done only for the purpose of preprocessing and does not affect the actual language grammar used in the output at all. All preprocessing instructions are transformed in one way or another into codes acceptable to standard C. The resultant output will be compiled as usual to allow any run-time checks to be done and at the same time any unacceptable syntax will be detected and reported as an error.

As it is not possible to cater for all the possible locations of comments in C, we need to be able to differentiate between ordinary C comments and preprocessing comments. All preprocessing instructions inside C comments will have the prefix 'PP' before any preprocessing instructions to differentiate them from ordinary C comments. Only preprocessing comment instructions will be processed further by the system. In this experimental system, preprocessing comments may be provided only in the following places :

i) in places where a statement inside a function can exist;

ii) external to a function;

iii) at the beginning of a scope;

iv) at the beginning of a declaration; and

v) before each declared variable.

Therefore, we make use of the comments by having the preprocessing instructions embedded in them so that only the right preprocessor can detect the appropriate instructions for it in the comments. A system for recognition is devised so that each preprocessing comment of a preprocessor can be recognised with a unique comment notation prefixed to the preprocessing instruction. In this way only the preprocessing comments that are prefixed for a particular preprocessor are processed while the rest are sent to the output.

Next, we consider the requirement to do preprocessing functions when preprocessing keywords representing preprocessing functions are recognised. This has been achieved through the use of Yacc. In Yacc, any codes to be executed when certain structures are recognised can be inserted as actions to be executed when parsing is done. The preprocessing function is done by inserting codes that do static checks or run-time checks through insertion and extension of text.

We now discuss how text can be inserted or expanded using the system. In the user specification file, a specification of a text expansion routine that defines a template of the text to be recognised in the Yacc actions section is specified together with a template of the extended text. The system then transforms the specifications of text insertion/expansion definition into automatically generated codes involving the handling of text which are inserted into the generated system Yacc specification file before it is being compiled and linked with others to produce the preprocessor.

Last on the list is the linkage of run-time codes for run-time checking. This is required especially if the run-time checks involve codes that have to be supplied by the preprocessor builder (user). It is best to bring together all the run-time codes of a preprocessor into one file which is then specified for linkage in the specification. When reading the user specification file, the system transforms the specification into codes that attach an **include** statement to the beginning of the output program so that the modified program is preceded by an **include** statement linking the run-time routine file.

The list, however, does not imply that these are the only requirements of the system. These we can consider as major requirements while other requirements will be discussed when and where appropriate in the later sections of this chapter.

In the next section we discuss the kinds of specifications that are required for the user to convey the characteristics of a preprocessor to the system.

3.2.2 : The Specification Language PSL

The instructions to the system concerning preprocessing function requirements is in a specification file of the intended preprocessor. To state the requirements of the functions that we require it to do, we need a specification language. In a way, the design and format of the specification language reveals to a certain extent, the capabilities of the system. An interface to help the user in specifying the preprocessor is discussed in Section 3.4.

Before we make specifications with reference to the system C productions, it is best that we define the structures that make up the grammar. A set of productions represents a grammar. A production has a non-terminal on the L.H.S. with a set of production-rules on the R.H.S.. A production-rule then is made up of symbol terms (terminals/non-terminals).

Let us now consider a requirement and its implementation step by step. Firstly, we would like to convey preprocessing instructions through standard C comments. The comments have to be specialised so

that they can be recognised only by the intended preprocessor. Therefore, preprocessing comment instructions of a particular preprocessor should be prefixed with a unique identifying notation. This requirement in specification has been transformed into the specification CNOTATION@. This means that if we specify

CNOTATION@TESTPRO ,

this instructs the system to build a preprocessor that will recognise standard C comments with preprocessing instructions preceded by the notation TESTPRO. Thus, only preprocessing comments of the form /*PPTESTPRO .....*/ will be recognised and processed. Note that @ is the specification delimiter in the specification file.

The next thing that we would like to be able to do is to introduce new keywords into the C language. This will allow the user to include the preprocessing keywords as keywords of the C language i.e. only for the purpose of preprocessing. The specification taken to convey this instruction is KEYWORD@. For example,

KEYWORD@KEYWD1

Repeated sequences of keyword definitions can be specified one after another on separate lines ending with the specification delimiter. To instruct the system not to print a keyword to the output, we postfix that

keyword with the string '(NP)'.

Next we consider how to add new actions to a production-rule. This specification is required so that when parsing is done during preprocessing, various preprocessing codes can be executed and processed including input text insertion and expansion. To insert an action into a production-rule for a particular symbol term, we need to furnish the system with the production number, production-rule number and symbol term number. Numbers referred to are with reference to the CPBS C Language Productions listed in Appendix II. The format in specification is as follows:

```
INS_ACTION
PROD@.....
RULE@.....
ACT@.....
@{.....
.....
.....
}
@
```

The production-rule number specification together with the action number (symbol term number) can be repeated if there is more than one production-rule to have actions inserted or more than one action to be

inserted on a single production-rule.

The internal structure carrying a symbol term is known as a SYMSTRUC. Thus, SYMSTRUC(1) is the structure carrying the first symbol term in a production-rule while SYMNM(2) refers to the string name of the second symbol term of a production-rule. Note that actions must be enclosed in a combination of a curly bracket and the specification delimiter; and that there should be no blank lines in them.

Apart from inserting further actions for existing production-rules , the is also a need to extend the language during preprocessing by adding new production-rules containing preprocessing keywords. This is done using the specification keyword INS_PRODRULE. This specification is quite similar to the previous one except that in this case we first specify a new production-rule before inserting actions (if any). The format for inserting a new production-rule is :

```
INS_PRODRULE
PROD@.....
PRULE
@.....
@
ACT@.....
@{.....
```

.....

}

@

In a production-rule, to instruct the system not to print a symbol term such as a preprocessing keyword to the output, we postfix that symbol term with the string '(NP)'.

A further modification that can be made to the grammar of C for the purpose of preprocessing is to add a new production with a set of production-rules including actions. A complete production includes a L.H.S. non-terminal with production rules on the R.H.S.. The specification format is similar to INS_PRODRULE but here no production number needs to be specified. An additional item to be specified is the non-terminal on the L.H.S.. The specification format is as follows :

ADD_NEWPROD

LHS@.....

PRULE

@.....

@

ACT@....

@{.....

.....

}

@

Repetitive specification of PRULE and ACT is allowed as in the previous specification for inserting actions.

Having considered modifications that can be made to the grammar of C for the purpose of preprocessing, we now consider specifying text inclusion and expansion to the system. The specification that has been implemented offers a number of text expansion variables. SYM refers to the current symbol corresponding to the action which contains the required expansion. STRx is a set of string expansion variables for which STR1 is a member and VALx is the set of integer expansion variables. The format of the specification is as follows :

TEXT_EXP

FN_HEAD@.....

EX_TEXT@.....

The instruction to expand a certain structure of the C input program or to insert additional codes will be given in an action of a symbol term. TEXT_EXP is in fact a declaration of a text expansion routine that specifies a definition in FN_HEAD with a template of the extended text in EX_TEXT. Note that any character strings other than the stated

expansion variables will be printed as in the specification. The convention used in between double quotes in a printf statement is also required here i.e. for example a newline (\n) in the EX_TEXT section will be written as \\n.

Another useful facility is prefixing. In order to avoid name clashes between variables of the input program and those of the run-time checking routines, we prefix variables in the input program. For example, to avoid name clashes between variables declared in a checking routine where the variables are already prefixed with the string "sfx" to create uniformity, the variable names of the input program which begin with the string "sfx" are prefixed with another "sfx" so that any name clash is avoided. The specification is implemented in the form

PREFIX@.....@WITH@..... .

To extend the language by adding new productions, we need to be able to declare new non-terminals to be on the L.H.S. of a new production. This is done by specifying one or more non-terminals as follows :

NON_TERMINAL

@.....

@

We now consider the specification facility required to allow run-time checks to be done by run-time routines. This requires some sort of specification that can link a file containing run-time check routines in the output of the preprocessor. The facility has been implemented and the keyword RUNTIME_FILE@ is used for this purpose in the specification file.

A specification keyword is also required to specify any file/files that may contain variable declarations and routines that are used in the inserted actions in the productions. This specification is given through the keyword SYNT_INCFILE@. One or more files can be specified using this keyword each separated from the next with a comma.

A specification language has been built to cater for the specification requirements mentioned in this section. A full definition of the specification language is given in Appendix I. Examples of preprocessors specified using the specification language are discussed in Chapter 4 and 5. A manual for the user is also made available in Appendix V.

3.2.3 : Implementation

The preprocessor building process is divided into various stages. The stages all represent translation processes from the reading of the user specification file through to the linking up of object codes to produce the required preprocessor. The basic preprocessor is made up of two

specification files, and a few definition and routine files. The lex specification file **pplfile** and the Yacc specification file **ppyfile** when processed through their respective tools, create a preprocessor that gives an output similar to its input. These files are supplied with standard routines to provide the basic processing of the input to the output. Listings of CPBS have been prepared by the author and are available from him. This includes a **ppjoin** routine that links lexemes representing constructs of the language together in a linked-list to form the resultant output program after parsing has taken place.

Next we describe the format of the files **pplfile** and **ppyfile**. Both the files have lines prefixed with an alphanumeric field to help with merging and sorting. In **ppyfile** each non-terminal/terminal in a production is placed one per line with an empty action for every symbol term on the R.H.S.. A section of the formatted file representing a production is given below :

```
s01.s00s00s00@primary_expr
s01.s01s01s00@: identifier {
s01.s01s01x99@}
s01.s02s01s00@| CONSTANT {
s01.s02s01x99@}
s01.s03s01s00@| STRING_LITERAL {
s01.s03s01x99@}
s01.s04s01s00@| LP {
```

```
s01.s04s01x99@}
s01.s04s02s00@expr {
s01.s04s02x99@}
s01.s04s03s00@RP {
s01.s04s03x98@ J3;
s01.s04s03x99@}
s01.x99x99x99@;
```

This format is taken to ease the automatic generation process involving the insertion of actions, production-rules and new productions. The system will merge these files with their corresponding files generated from the specification in the specification file. This allows the proper grouping of the various sections of the resultant Lex and Yacc specification files. A more elaborate form of processing is required for the resultant Yacc specification file for modifications due to insertions and expansions. For example, a new inserted action would mean that the parameters of the **ppjoin** routine call mentioned above have to be changed because the additional action in Yacc counts as a symbol term. At this initial stage i.e. before merging and the insertion of actions/productions, it is not possible to fix the parameters of each **ppjoin** routine call.

The implementation of the system begins with the first stage of the translation i.e. reading in the specifications and translating them. This is done through an Awk program **ppawk** which translates the entire specification into specifications for Lex and Yacc into files **lexfile** and **yaccfile** respectively together with some auxiliary files.

Basically, the Awk program is written to do the following :

a) translate the comment notation specification in the user specification file into lex specifications that allows the special notation specified to be recognised by the eventual preprocessor;

b) transform the inclusion specification of the keyword SYNT_INCFILE@ into the appropriate file for insertion into the appropriate section;

c) translate the KEYWORD@ specification in the specification into specifications of Lex into the appropriate sections;

d) transform any prefixing instruction into Lex specifications with codes to do prefixing;

e) translate the non-terminal specification into a specification of Yacc declaring the non-terminals;

f) transform the INS_ACTION keyword specification into one suitable for Yacc with the appropriate fields so that the inserted actions are rearranged into the correct positions in the resultant Yacc file after merging and sorting;

g) translate the INS_PRODRULE specification. This involves analysing each new production-rule to provide the correct parameter to the **ppjoin** routine and also to consider symbol terms which are prefixed with (NP) which are not to be printed;

h) translate the ADD_NEWPROD specification involving the proper insertion of new productions with routines similar to (g);

i) translate the TEXT_EXP keyword specification to implement insertion and expansion of text. This involves scanning the specification in EX_TEXT for text expansion variables and transforming the specification into an Awk program which processes the Yacc specification file by inserting automatically generated code implementing insertion/expansion in one of the later stages of preprocessor building;

j) transform the RUNTIME_FILE specification into codes inserted into the Yacc file that when executed gives an the output from the preprocessor preceded by an **include** statement linking the specified run-time file.

The next stage in preprocessor building is the merging and sorting of the contents of **pplfile** and **ppyfile** with the respective files **lexfile** and **yaccfile** derived from the user specification file by the **ppawk** Awk program. This is done simply by concatenating the respective files and piping the output into Unix system sort.

Once the merging and sorting have been done we then need to remove the prefixed field. This is the next stage. An Awk program **remove** does this by taking the input piped from the last stage, processing it (removing the field) and piping it to the next stage.

The next stage involves a modification of the resultant Yacc file piped from the previous stage. As discussed earlier with reference to the format of the Yacc files, this stage involves the removal of empty actions. This is done through an Awk program **empact**.

A further stage handles the expansion to the **ppjoin** routine call abbreviated in the initial files. It has been abbreviated to simplify the specification in Yacc and also to carry only with it the actual number of symbol-terms in a production-rule. There is no use expanding it in the initial stages as the parameters to the **ppjoin** routine call will probably be changed due to inserted actions. For example, abbreviation for ppjoin(SYMSTRUC(1),SYMSTRUC(2)) is given as J2 only. The expansion also involves the replacement of the the names SYMSTRUC and SYMNM. SYMSTRUC(X) is a positional variable in Yacc referring to

the structure representing the Xth. symbol term on the production-rule while SYMNM(X) represents an implementation field **name** of that structure which in turn represents the lexeme of the symbol term. This expansion is done through a lexical specification **lexpand.l** using Lex which replaces every occurrence of of the abbreviated ppjoin routine call and the names SYMSTRUC and SYMNM. Note that at this stage, the J calls are not only from the original formatted file **ppyfile** but also from the transformed specification read from the user specification file.

This next stage processes the **ppjoin** routine call further. It makes an alteration to the calling parameters of that routine due to the insertions of new actions. For example, for a simple production-rule as

```
| symb1 symb2 symb3
```

the call will be ppjoin(SYMSTRUC(1),ppjoin(SYMSTRUC(2),SYMSTRUC(3))) but an additional action for symbol term 2 inserted between symbol term 2 and 3 will force a change in the routine call to ppjoin(SYMSTRUC(1),ppjoin(SYMSTRUC(2),SYMSTRUC(4))). This alteration is achieved through an Awk program **change** that works out how many symbol terms actually exist in each production-rule and keeps track of the positions of these symbol terms. It also takes into consideration the (NP) prefixed to certain symbol terms not to be printed to the output, by not passing their positional variables.

The last stage is the execution of the text expansion Awk program **awkins** created during the first stage, replacing and inserting all text expansion calls into the piped yacc specification file which is finally redirected into a final yacc spec file.

All these stages in the building of a preprocessor are coordinated through a shell script which passes the output of one stage as the input of another through the Unix piping facility. Given below is the shell script **psystem** that puts together the stages into one preprocessor building system.

```
awk -f ppawk $1
cat pplfile lexfile|sort -n|remove.s>$2'.lf'
cat ppyfile yaccfile|sort -n|remove.s|empact.s||expand|change.s|awkins.s>$2'.yf'
yacc -d $2'.yf'
lex $2'.lf'
cc -o $2'.pp' lex.yy.c y.tab.c  main.c -ll -ly
echo '/lib/cpp -P -C $1|' $2'.pp'>$2
chmod u+x $2
rm lexfile yaccfile
```

In the process of creating the required preprocessor, the system creates auxiliary files to assist it in the generation of the preprocessor. Most of the files will be deleted on completion of preprocessor except two

files with the extensions **.lf** and **.yf**. Both of these files remain so that the user can check on the underlying structures of the lexical and syntactical stages.

It should also be noted that the input to any preprocessor is preprocessed first by the standard C preprocessor (CPP) to eliminate standard C preprocessing statement.

3.3 : Using the System

To use the system, the user has to prepare a user specification file of the intended preprocessor using the specification language and with reference to the system C productions. To build a preprocessor **prepro** using a specification file **specfile**, a user types the command

psystem specfile prepro .

and the resultant preprocessor is executed with an input program **input_prog** with the command

prepro input_prog .

There are times when new productions or production-rules introduced by the user can cause ambiguity and conflicts. However, this does not mean that a parser for the preprocessor will not be produced. The system invokes two disambiguating rules by default :

i) in a shift/reduce conflict, the default is to do the shift;

ii) in a reduce/reduce conflict, the default is to reduce by the earlier grammar rule.

Therefore, the system will still produce preprocessors even in the presence of conflicts. Messages about those conflicts will be reported to the user by the system. For further information and to understand further how the system handles ambiguities, conflicts and errors through Yacc, refer to the CPBS User Manual in Appendix V.

3.4 : The Interface

A user friendly interface is available to the user in specifying the preprocessor. The user interface prompts the user for details to be specified which would otherwise have to be presented in a particular specification language format using special specification keywords. The interface actually builds the specification file as the dialogue progresses.

To use the interface a user initiates a dialogue session by typing the command **intface**. The specification will be written into specification file **sfile**. **sfile** can then be used as input to the system as explained in the

previous section.

An example of a dialogue session is given in Appendix VII. In the next chapter we discuss the capabilities and limitations of the system as a preprocessor builder.

CHAPTER FOUR

THE SYSTEM AS A TOOL

4.1 : What Is On Offer?

After discussing the design and implementation aspects of the preprocessor builder in the last chapter, we take another look at the system from a different viewpoint, that of a user. How useful is this tool that was built? What has it got to offer to the user?

A relatively simple and easily used system was built to simplify and ease an otherwise substantial task of building preprocessors. It reduces the work of building a preprocessor to the task of merely specifying it to the system using a simple but sufficient specification language. It is an automated system that transforms a set of specifications into the required preprocessor, permitting the building of C language checking and extending preprocessors through the simple specification procedure. Basically, it offers the user a dynamic tool that comprises of a specification modifiable lexical and syntax analysers, facilities for text replacement, expansion and insertion; and run-time linkage.

In short, the system should be able to build any preprocessor that can be built through programming using the C language. The main factor demonstrated is simplicity through automation. Thus, allowing the user

to concentrate only on specifying the actual preprocessing that is required directly with reference to the grammar that represents the structures of the language. There is actually no need to master other tools as the system comes as a complete package with a specification language, basic preprocessing routines, automation facilities together with the compilation and linkage steps leading to the eventual preprocessor. Another point to note is that, as it does not directly involve machine dependent instructions, it is also portable and can be transported to any machine where the standard language development tools Lex and Yacc are present.

Another point to note is that all preprocessors built using this system allow preprocessing instructions of other preprocessors to pass through them without being detected. Generated preprocessors can also be piped together to do several different preprocessing tasks consecutively.

Let us consider a short example in introducing a new language construct into C to demonstrate the simplicity of using the system. The task at hand is to introduce the familiar **repeat-until** construct of the Algol-like languages into C. We therefore need a preprocessor that can accept the **repeat-until** construct. A new construct to be implemented through preprocessing necessitates a mapping of it back down into the base language i.e. the standard C language. We therefore map into the **do-while** construct of C which is similar in form. We could alternatively use a labelled **if** statement and a **goto** statement. Given

next is a specification of the preprocessor to be implemented.

```
CNOTATION@RP            /*  comment notation */

KEYWORD                 /* specifying keywords */
@REPEAT
UNTIL
@

NON_TERMINAL            /* specifying a non-terminal */
@rpuntil
@

INS_PRODRULE            /* inserting production-rule
PROD@61                 into existing production */
PRULE
@rpuntil
@

ADD_NEWPROD             /* adding a production */
LHS@rpuntil
PRULE
@REPEAT statement UNTIL LP expr RP SC
@
ACT@1                   /* inserting action for a
@{                      particular symbol_term */
TS_EX1(SYMSTRUC(1));    /* an expansion function call */
}
@
ACT@3
@{
TS_EX2(SYMSTRUC(3));
}
@
ACT@5
@{
TS_EX3(SYMSTRUC(5));
}

SYNT_INCFILE@var.h      /* specifying file  to support
                        preprocessing      */
TEXT_EXP                /* expansion functions templates */
FN_HEAD@TS_EX1(SYM)
EX_TEXT@do
FN_HEAD@TS_EX2(SYM)
EX_TEXT@while
FN_HEAD@TS_EX3(SYM)
EX_TEXT@!(SYM)
```

It can be argued that the version of the **repeat_until** construct could be implemented using the macro facility of the CPP but note that it cannot be implemented properly using the **if-goto** combination. This is because CPP cannot be used satisfactorily to generate the necessary labels.

However, this can be properly done using CPBS. Furthermore, no problems arise when the construct is nested. Clashes between label names of the input programs and those inserted by the preprocessor can be handled through the use of the prefixing facility of the system. Furthermore, macro substitution through the CPP does not allow for proper syntax checking to be done. For example, a syntax error in the form of a missing **repeat** lexeme will pass through the preprocessing stage undetected only to be detected at a later stage at compile-time.

Another simple example is the implementation of a user specified debugger through the use of a preprocessor with user supplied debugging routines where debugging is only done whenever an intended program is passed through the debugging preprocessor before compilation. Here, a simple specification file with specifications of the comment notation and a run-time file reference are all that is required. In a sense, it is an implementation of conditional compilation as only by passing the intended program through the debugging preprocessor will the debugging codes or calls to those codes be processed by the compiler.

4.2 : Example I : A Pre- And Post-Condition Preprocessor

4.2.1 : The Requirement

In this section we look at a pre- and post-condition preprocessor that was built using the system. The system was used to provide insertions into the input programs in the form of calls to pre- and post- condition check routines and to allow those user supplied routines to be linked to the output programs. The pre condition check call was to be inserted right after the first declaration section in the intended function i.e. before any executable statement, while the post condition check call was to be inserted before the **return** statement.

The pre condition call to the routine carried the values passed to the formal parameters of the function. This was to check whether the actual parameters to the function call carried values that satisfy its pre condition. In the case of the post condition, it also included the value returned by the function : a check on whether the function did what it ought to and returned the expected result. A standard notation was used such that the pre- and post-condition user routines are named with either the string 'pre-' or 'post-' prefixed before the function name. This was to help in the automation process.

4.2.2 : The Mechanism of Implementation

From looking at the CPBS C language productions in Appendix II, we observe that the insertions involved productions [54] on the second ([54].2) and fourth ([54].4) production-rules; and production [62] with production-rule 5 ([62].5). Production-rule [38].7 gives the name of the function to be stored. Scoping levels have to be monitored to allow insertion only on the outermost scope of a function. This was achieved through fixing a level counter at production-rules [55].1 and [56].1.

A preprocessing instruction FNCHECK was to be to inserted before a function definition in an input program to instruct pre and post condition checks to be done. This involved inserting two new production-rules into production [67] involving function definitions by setting up a check requirement flag.

4.2.3 : The Specification File

Declarations of variables used in the actions was declared in file **ppcdef.h** and the name of file containing the checking routines was **ppcheck.h**. Given below is the specification file required for the task of building the required preprocessor.

```
CNOTATION@COND          /* comment notation */

KEYWORD                 /* specifying keyword FNCHECK */
@FNCHECK(NP)
```

```
@

INS_ACTION
PROD@38
RULES@7
ACT@4
@{
if (checkflag==1)
        {temp1=SYMNM(1);  /*store required details */
         temp2=SYMNM(2);
         tstrptr=pplinkup(SYMSTRUC(3));
         thisptr=anarr;
         sprintf(thisptr,"%s%s%s",temp1,temp2,tstrptr);
        }
}
@

INS_ACTION
PROD@54
RULE@2
ACT@1
@{                  /* at the beginning of a function scope */
if ((levcount==1) && (checkflag==1))
        EXPAND_PRE(SYMSTRUC(1),thisptr);
}
@
RULE@4
ACT@2
@{                  /* at the beginning of a function scope */
if ((levcount==1) && (checkflag==1))
        EXPAND_PRE(SYMSTRUC(2),thisptr);
}
@

INS_ACTION
PROD@55
RULE@1
ACT@1
@{
++levcount;     /* level counter : level 0, out of function*/
}
@

INS_ACTION
PROD@56
RULE@1
ACT@1
@{
--levcount;
}
@

INS_ACTION
PROD@62
```

```
RULE@5
ACT@3
@{
if (checkflag==1)  /* expanding return statement */
        EXPAND_POST(SYMSTRUC(1),SYMNM(2),thisptr);
}
@

INS_PRODRULE
PROD@67
PRULE           /* processing FNCHECK keyword */
@FNCHECK(NP) declarator function_body
@
ACT@1
@{
checkflag=1;  /* do pre- post- checks */
}
@
ACT@3
@{
checkflag=0;
}
@
PRULE       /* processes the FNCHECK keyword */
@FNCHECK(NP) declaration_specifiers declarator function_body
@
ACT@1
@{
checkflag=1;
}
@
ACT@4
@{
checkflag=0;
}
@

SYNT_INCFILE@ppcdef.h

RUNTIME_FILE@ppcheck.h

TEXT_EXP      /* text expansion templates */
FN_HEAD@EXPAND_PRE(SYM,STR1)
EX_TEXT@SYM \\n pre_STR1);
FN_HEAD@EXPAND_POST(SYM,STR1,STR2)
EX_TEXT@post_STR2,STR1);\\n SYM
```

This preprocessor has been used as an example for the manual in Appendix V. A sample of the input and output programs to the preprocessor are given in the manual.

4.3 : Example II : The Index Checker

Sec. 4.3.1 : The Requirement

Next we look at the requirements of a preprocessor that inserts statements to perform run-time checks on the validity of indices in array expressions. Basically, what is required is an index checker that accepts as input a C program and outputs an instrumented program that when compiled and executed will print out error messages on the VDU giving information pertaining to i) name of array involved; ii) the out of bounds index; and iii) its line number in the original program.

We would like to check on array expressions of the following type :

i) one-dimensional array;

ii) multi-dimensional array;

iii) nested array.

The user will have an option of the following :

i) full index checking on all arrays;

ii) index checking on specified arrays only;

iii) index checking at specified on/off points; and

iv) index checking in specified scopes.

4.3.2 : Mechanism of Implementation

The basic structure that is to be checked is the array index of an array expression. To check whether an index is out of bounds, we obviously need to know the size of the array which is declared in an array declaration.

With reference to the system list of C productions, the production representing array expressions is production-rule [2].2 while production-rule [38].4 represents array declarations. In production-rule [2].2 **postfix-expr** can be recursively reduced to **primary-expr** by production-rule [2].1 and reduced further to **identifier** by production-rule [1].1. In production-rule [38].4, **declarator2** can be recursively reduced to **identifier** by production-rule [38].1.

Note that a multi-dimensional array can be obtained by recursive reduction on **postfix-expr** in production-rule [2].2. Nested array indices can be obtained by reduction from production [20] through to production [2] skipping production [3], [5] and [19].

On recognising an expression of the form

```
a[b]=c;
```

we want to check the index by expanding it to

```
a[(sfx[1]=b)<0||sfx[1]>=arrsize_a?error(sfx[1],lineno,"b"):sfx[1]]=c;
```

and an expression of the form

```
a[b[c]]=d;
```

is expanded to

```
a[(sfx[2]=b[(sfx[1]=c)<0||
 sfx[1]>=arrsize_b?sfxxerror(sfx[1],lineno,"b"):sfx[1]])<0||
        sfx[2]>=arrsize_a?sfxxerror(sfx[2],lineno,"a"):sfx[2]]=d;
```

On recognising an array declaration we want to store the name of the array and its dimensions in the symbol-tables. During parsing, the input program is parsed enabling declared information to be stored and array expressions to be checked in the process. Expressions that consist of array expressions to be modified are modified accordingly and stored in the linked-list which is printed out once the whole program has been parsed.

Scoping is maintained throughout with the help of two globally declared linked-lists. One is used to store externally declared array variables and the other to store array variables declared inside functions. For every scoping level in a function, there exist a sub-list. The inner level list is joined to the end of the outer level list and later deleted from the function list when coming out of the inner level.

To discuss the storage of information from array declarations there are two types of array declarations to be considered : i) globally declared array variables (outside functions); and ii) array variables declared inside functions (eg. auto variables).

i) Globally declared array variables.

Through studying the system productions of C, if there is an array variable declared globally, the reduction route will definitely go through production-rule [65].2 and production-rule [38].4. Therefore, we check at production-rule [65].2 whether an array declaration has been declared through it. The easiest way is to create a flag say **arrflag** which is checked after production-rule [65].2 has been passed. We require a temporary list to store the information while at production [38] and then link it to the appropriate linked-list after coming out from production-rule [38].4 back to production-rule [65].2. Care is taken in setting up flags and their initialisation.

ii) Array variables declared inside functions

A study of the grammar productions reveals that production-rule [54].4 is to be considered when considering array variables declared inside functions. The use of this production-rule simplifies the problem of implementing scoping considerably. As in (i), a check is made to check if an array has been declared through non-terminal **declaration_list** in

production-rule [54].4. If it is true the temporary list formed at production-rule [38].4 is linked to the external variable list. A scope pointer is assigned to the head of the temporary list before it is linked. This is to assist in deleting the sub-list when coming out of a scope level.

At production-rule [38].4 the declared information is stored. On completion of the production-rule, a new item with information obtained from the attributes of the production-rule is linked to the end of the existing temporary list. With reference to the attributes of the production, we can check whether an array used in an expression is in the two global lists and if they are, provide run-time checks by expanding the indices with checking statements. Care has been taken to consider multi-dimensional arrays resulting from recursive self-referencing.

The structure of an entry in the symbol-table is

```
struct ENTRY
        { char arrname[ARRNMSIZE];
          int dimsize[MAXDIM+1]
          struct ENTRY *next,*prev;
        }
```

To avoid any name clashes occurring between the variable names in the input program and the suffix-checking array variable sfx in the run-time routine, the PREFIX@sfx@WITH@sfx is given. In order that we

can specify various preprocessing commands, it is necessary that we introduce those commands using the KEYWORD specification. Preprocessing instructions will be in the form of specialised comments which are valid C language comments. The specification CNOTATION is used to specify instructive comments preceded by SUF. The lexical analyser recognises such specialised comments, strips them off to reveal the instructive keywords and passes them on to the parser. The preprocessing comments are :

i) /*PPSUF COM_CHECK */

instructs full checking from the time this instruction is read;

ii) /*PPSUF SCOPE_CHECK */

instructs checking only in the scope containing this instruction i.e. between two matching curly brackets;

iii) /*PPSUF CHECK_ON */

instructs checking on just one specific array from the time it is declared till the time it diminishes through scoping;

iv) /*PPSUF SINGL_ARR */

instructs checking on just one array at specific on/off points indicated by (vii) and (vii);

v) /*PPSUF ALL_ARR */

instructs checking of all arrays declared from the point of instruction at specified on/off points as in (iv);

vi) /*PPSUF START_CHECK */

represents starting point to begin suffix-checking;

vii) /*PPSUF END_CHECK */

represents ending point to stop suffix-checking.

4.3.3 : The Specification File

This section gives the specification file required to generate the index checker. Implementation variables and routines together with run-time routines specified in the specification file are listed in Appendix IV. The specification file is as follows :

```
CNOTATION@SUF

KEYWORD
@START_CHECK(NP)
END_CHECK(NP)
ALL_ARR(NP)
SINGL_ARR(NP)
COM_CHECK(NP)
SCOPE_CHECK(NP)
CHECK_ON(NP)
@

PREFIX@sfx@WITH@sfx

NON_TERMINAL
@aarpre
com_stat1
@
```

```
INS_ACTION
PROD@2
RULE@2
ACT@2
@{
 if (closebrac==0)
          push(1);
  else
    ++(top->val);
 closebrac=0;
 }
@
ACT@4
@{ if (closebrac==1)
                    pop();
              if (checkflag==1)
               {
                 k=thesize(SYMNM(1),top->val);
              if (k>0)
               { ++t;
                 EXPAND_SUF(SYMSTRUC(3),SYMNM(1),t,k,lcount);
               }
              }
              closebrac=1;
}
@

INS_ACTION
PROD@4
RULE@1
ACT@1
@{
closebrac=0;
}
@

INS_ACTION
PROD@37
RULE@1
ACT@1
@{
dimcount=0;notproper=0;condfn();
}
@
RULE@2
ACT@2
@{
dimcount=0;notproper=0;condfn();
}
@

INS_ACTION
PROD@38
RULE@2
```

```
ACT@3
@{
notproper=1;arrflag[levcount]=0;
}
@
RULE@3
ACT@3
@{
notproper=1;arrflag[levcount]=0;
}
@
RULE@4
ACT@4
@{
                    if ((notproper==0) && ((storeflag==1)||(sigflag==1)))
                     {
                      arrflag[levcount]=1;++dimcount;
                      if (dimcount==1) temp_entry=store(SYMNM(1));
                        temp_entry->dimsize[dimcount]=atoi(SYMNM(3));
                      }
 }
@
RULE@6
ACT@4
@{
notproper=1;arrflag[levcount]=0;
}
@
RULE@7
ACT@4
@{
notproper=1;arrflag[levcount]=0;
}
@

INS_ACTION
PROD@55
RULE@1
ACT@1
@{
fnflag=1;++levcount;ptrflag[levcount]=0;
}
@

INS_ACTION
PROD@56
RULE@1
ACT@1
@{
        if (ptrflag[levcount]==1)
                delete(ptr[levcount]);
        --levcount;
        if (levcount==0)
                {
```

```
                    fnflag=0;tc=(-1);
                    notproper=0;dimcount=0;storeflag=0;
                    checkflag=0;sigflag=0;t=0;closebrac=0;
                   }
 }
@

INS_ACTION
PROD@59
RULE@2
ACT@2
@{t=0; }
@

INS_PRODRULE
PROD@38
PRULE
@com_stat1(NP) declarator2 LB constant_expr RB
@
ACT@5
@{
                    if (notproper==0)
                     {
                      arrflag[levcount]=1;++dimcount;
                      if (dimcount==1) temp_entry=store(SYMNM(2));
                       temp_entry->dimsize[dimcount]=atoi(SYMNM(4));
                      }
}
@
PROD@52
PRULE
@START_CHECK
@
ACT@1
@{
checkflag=1;
}
@
PRULE
@END_CHECK
@
ACT@1
@{
checkflag=0;storeflag=0;
}
@
PROD@54
PRULE
@begin SCOPE_CHECK(NP) declaration_list statement_list end
@
ACT@2
@{
storeflag=1;checkflag=1;
}
```

```
@
ACT@5
@{
storeflag=0;checkflag=0;
}
@
PRULE
@begin aarpre(NP) statement_list end
@
PRULE
@begin aarpre(NP) declaration_list end
@
PRULE
@begin aarpre(NP) declaration_list statement_list end
@

INS_PRODRULE
PROD@65
PRULE
@COM_CHECK(NP)
@
ACT@1
@{storeflag=1;checkflag=1;
}
@

INS_NEWPROD
LHS@com_stat1
PRULE
@SINGL_ARR
@
ACT@1
@{sigflag=1;
}
@
PRULE
@CHECK_ON
@
ACT@1
@{sigflag=1;checkflag=1;
}
@
LHS@aarpre
PRULE
@ALL_ARR
@
ACT@1
@{storeflag=1;
}
@

SYNT_INCFILE@sfxstruc.h,sfxstack.h,sfxdef.h

RUNTIME_FILE@suf.rtm
```

```
TEXT_EXP
FN_HEAD@EXPAND_SUF(SYM,STR1,VAL1,VAL2,VAL3)
EX_TEXT@(sfx[VAL1]=SYM)<0 ||\\n\\t sfx[VAL1]>=VAL2?
     sfxxerror(sfx[VAL1],VAL3,\\\"STR1\\\"):sfx[VAL1]
```

4.4 : Limitations of the System

One of the limitations of the system is that the resultant grammar that is finally presented to Yacc by the system to be processed must be LR. Another is that, as it is a preprocessor builder for C, it can only build preprocessors that preprocess an input program producing an output in standard C. This rules out any possibility of producing an output other than that that can be accepted by the C compiler and therefore tasks that require the system to interface with another system as in parallel processing for example, cannot be accommodated.

An important overall limitation is that a preprocessor cannot introduce any new type-checking since this is a compile time task. For the same reason, it does not appear feasible to implement an exception mechanism using the system.

Apart from the limitations above, the system should be capable of building any preprocessor that can possibly be built through programming using the C language. However, in the context of language extensions, we are therefore limited to the pre-compilation stage through the use of the preprocessors. It should be noted that the extreme case of language extension that can possibly be implemented through the use of a

preprocessor is one that will have to finally map down into the standard structures of C. In the next chapter we look at a final example of the system at work.

CHAPTER FIVE

A FINAL EXAMPLE : GENERIC PACKAGES IN C

5.1 : Introduction

This chapter demonstrates the system with yet another example - a preprocessor that supports generic packages. The aim of this chapter is actually two-fold. First, to demonstrate further the effective use of the completed system. Secondly, to present a quite simple but effective method of grafting generic packages into the C language. The example given is not an elaborate one since details of the system facilities available can be demonstrated effectively using uncomplicated concise examples.

5.2 : Generic Packages

A package in C is an information hiding and data encapsulation mechanism. It can be used to group logically related entities such as constants, variables and functions. There is a visible and private part of the package. Only the entities specified in the visible part of a package specification can be referred to by the package user.

Generic packages are packages that can accept types as parameters. They act as a template for the declaration of true packages. A generic package must be instantiated to create a new package and more than one instantiation per generic package is possible. Generic packages provide examples of software reusability. With a generic package, packages involving similarly identical functions and routines need not be written over and over again. Programs become more manageable and program correctness will be much easier to check. Generic packages provides an excellent example of abstraction. Using a package, visible components can be declared for external reference while implementation aspects of the package can remain hidden from outside the package. In fact, a generic package is a stage further into abstraction. Also, through the use of generic packages changes to the types of variables used as parameters to a package will just be a single change during instantiation of that type.

5.3 : Generic Packages in C

The technique presented here permits the use of generic packages in C to provide a means for abstraction and genericity. Although it is not possible to emulate fully the concept of generic packages as in Ada, the work described in this chapter provides a C flavoured generic package that maintains the style of C while introducing a whole new concept. The generic packages have been implemented with the use of the preprocessor building system including the standard C Preprocessor (CPP). It is thought that the use of the CPP would simplify the

complexity of converting the package keywords to a minimum without too much confusion to the user, for example, some keyword expansions have been left to the CPP up to the stage before compilation to improve readability of the output program.

The usage of generic packages in C is represented by three component files :

i) a file **genpkg_X** defining a generic package usually written by a package developer;

ii) a file **inst_X.h** for instantiation of the generic package written by either the package developer or the user; and

iii) a calling program file that links itself to the newly created package instantiation, written by the user.

5.4 Implementation

To simplify the use of the generic packages and to mask the actual C codes involved when supporting them, a set of keywords were used. These keywords translate the requirements specified by the user through them into ordinary C codes. The translation process is done through the

expansion and insertion facility of the system and the macro substitution of the CPP. There are macro expansions which can only be done using the system, for example, the concatenation of two macro parameters.

The emulation of a package is done through the use of pointers to functions in C structures because in C structures can be defined containing pointers to functions as components while genericity is achieved through the process of macro substitution using the system and CPP. The hidden section of a generic package is represented by a **static** structure **private** containing components that are hidden from outside the package file due to the use of storage class **static.** Abstraction is thus achieved as the components of the hidden structure can only be referred to from within the generic package implementation file **genpkg_X.** All functions in the generic packages are declared as **static** and only those entities available as components of the visible package structure can be referred to from outside the package and only if the proper link has been made.

An example of a generic package **genpkg_STACK** is given below :

```
GENERIC
   CONST(STACK_SIZE,int)
   TYPE(STACK_TYPE)

G_PACKAGE(STACK)

VIS
FUNC(STACK_TYPE,pop);
FUNC(int,push);
FUNC(bool,empty_stack);
FUNC(bool,full_stack);
```

```
STACK_TYPE top_of_stack;
END_VIS(STACK)

PRIV
STACK_TYPE selem[STACK_SIZE];
int stacktop;
END_PRIV(STACK)

END_G_PACKAGE(STACK)

static bool full_stack()
{
return(private.stacktop==STACK_SIZE);
}

static bool empty_stack()
{
return(private.stacktop==0);
}

static STACK_TYPE pop()
{
 if (private.stacktop>0)
  {if (private.stacktop>1)
       STACK.top_of_stack=private.selem[private.stacktop-2];
   return(private.selem[--private.stacktop]);
  }
 else
  printf("error : stack underflow **********\n");
}

static int push(atype)
STACK_TYPE atype;
{
 if (private.stacktop<STACK_SIZE)
   {STACK.top_of_stack=atype;
    private.selem[private.stacktop++]=atype;
   }
 else
  printf("error : stack underflow **********\n");
}

PRIV_INIT
private.stacktop=0;
END

EXPPKG(STACK)
EXP(STACK,push);
EXP(STACK,empty_stack);
EXP(STACK,full_stack);
EXP(STACK,pop);
END
```

A generic package begins with a declaration of the generic parameters of the package using the keyword GENERIC. The keyword CONST specifies a constant variable of a particular type so that CONST(X,Y) specifies a variable X of type Y. The keyword TYPE specifies a generic type. Another keyword to be used under GENERIC is the keyword FN. FN(A,B) specifies a generic function A for use in function B which must be a component of the package. The GENERIC section of the package definition actually serves as a specification to the user concerning the characteristics of the generic parameters of the package.

The actual definition of the generic package begins with the keyword G_PACKAGE. The VIS keyword heads the list of visible components of the package. Keyword FUNC defines a function that returns a value of a particular type while the keyword PROC defines a function of type **void.** Therefore we have FUNC(X,Y) defining a function Y that returns a value of type X while PROC(T) defines a function T that does not return any value. Keyword END_VIS is the keyword to be used to signify the end of the visible section of the package.

The definition in the hidden section of the package begins with keyword PRIV and ends with the keyword END_PRIV. Apart from the package definition, variables and function definitions, the keyword PRIV_INIT declares a section in the package file to do any necessary initialisation of the private components. This section ends with the keyword END.

In order to allow and prepare a generic package for external use (export), an interface is defined through the use of keywords EXPPKG, EXP, and END as shown in the above given example.

To instantiate a generic package, a instantiation file **inst_X.h** is required. Instantiation of the generic parameters is done using instantiation keywords. An example of an instantiation file **inst_CSTACK.h** is given below :

```
PKGINST(STACK,CSTACK)
INST(STACK_SIZE,1000)
INST(STACK_TYPE,char)
PKGLINK(getCSTACK)
END_PKGINST(STACK)
```

Keyword PKGINST instantiates the package. PKGINST(X,Y) instantiates generic package X to create a new package Y. Keyword INST used in INST(U,V) instantiates entity U with V. PKGLINK(L) defines the function name L that will make the link to the newly created package. It is essential that this be defined for every single instantiation for proper package linkage. To denote the end of the instantiation process, END_PKGINST(X) specifies the end of instantiation of generic package X.

Once the new package has been created by the instantiation of the generic package, it can then be used just like an ordinary package. But for a program to actually use a newly created package from outside it,

the use of the package must be declared using some keyword instructions. Below is a program **Cmain.c** that links and makes use of a newly created package CSTACK.

```
/*PPGPKG

GETPKG(CSTACK)
FUNC(char,pop);
FUNC(int,push);
FUNC(bool,empty_stack);
FUNC(bool,full_stack);
char top_of_stack;
END_GETPKG(CSTACK)    */

main()
{
 FILE *ifp=fopen("INFILE","r");
 FILE *ofp=fopen("OUTFILE","w");

 /*PPGPKG getCSTACK(); */
 while (((*CSTACK.push)(getc(ifp)))!=EOF)
  {
   if (CSTACK.top_of_stack=='*' || (*CSTACK.full_stack)())
     while (((*CSTACK.empty_stack)())==FALSE)
        fprintf(ofp,"%c",(*CSTACK.pop)());
  }
 fprintf(ofp,"\n");
}
```

The use of a package necessitates a declaration in the calling program file of the intended package using the same keywords as those used to define the generic package except that the call from a using program begins with the keyword GETPKG and ends with the keyword END_GETPKG. Notice also that a call getSTACK() is made before any expression in the calling program to link the required package using the name specified in the instantiation file **inst_CSTACK.h**.

The system user specification file to provide the preprocessing required for preprocessing the instantiation file and the calling program is given below :

```
CNOTATION@GPKG

KEYWORD
@PKGINST
INST
PKGLINK
END_PKGINST
GETPKG
END_GETPKG
FUNC
PROC
@

NON_TERMINAL
@num_ident
any_type
instpkg
mainprog
@

INS_PRODRULE
PROD@65
PRULE
@instpkg
@
PRULE
@mainprog
@
PRULE
@any_type identifier SC
@

ADD_NEWPROD
LHS@instpkg
PRULE
@PKGINST LP(NP) identifier CM(NP) identifier RP(NP)
@
ACT@1
@{
GP_EX1(SYMSTRUC(1));
}
@
PRULE
@PKGLINK LP(NP) identifier RP(NP)
```

```
@
ACT@1
@{
GP_EX3(SYMSTRUC(1));
}
@
PRULE
@INST LP(NP) identifier CM(NP) num_ident RP(NP)
@
ACT@1
@{
GP_EX1(SYMSTRUC(1));
}
@
PRULE
@END_PKGINST LP(NP) identifier RP(NP)
@
ACT@1
@{
GP_EX4(SYMSTRUC(1));
}
@
ACT@3
@{
GP_EX8(SYMSTRUC(3));
}
@

ADD_NEWPROD
LHS@mainprog
PRULE
@GETPKG LP(NP) identifier RP(NP)
@
ACT@1
@{
GP_EX5(SYMSTRUC(1));
}
@
ACT@3
@{
GP_EX6(SYMSTRUC(3));
}
@
PRULE
@END_GETPKG LP(NP) identifier RP(NP)
@
ACT@1
@{
GP_EY1(SYMSTRUC(1));
}
@
ACT@3
@{
GP_EY2(SYMSTRUC(3));
```

```
}
@
PRULE
@FUNC LP any_type CM identifier RP SC
@
ACT@1
@{
GP_EX9(SYMSTRUC(1));
}
@

PRULE
@PROC LP identifier RP SC
@
ACT@1
@{
GP_EX9(SYMSTRUC(1));
}
@

ADD_NEWPROD
LHS@num_ident
PRULE
@identifier
@
PRULE
@CONSTANT
@
PRULE
@type_specifier
@

ADD_NEWPROD
LHS@any_type
PRULE
@type_specifier
@
PRULE
@identifier
@

SYNT_INCFILE@gpkg.h

RUNTIME_FILE@gen_pkg.h

TEXT_EXP
FN_HEAD@GP_EX1(SYM)
EX_TEXT@\n#define
FN_HEAD@GP_EX2(SYM)
EX_TEXT@SYM \n
FN_HEAD@GP_EX3(SYM)
EX_TEXT@\n#define LINK
FN_HEAD@GP_EX4(SYM)
EX_TEXT@\n#include
```

```
FN_HEAD@GP_EX5(SYM)
EX_TEXT@\nextern
FN_HEAD@GP_EX6(SYM)
EX_TEXT@struct { \n
FN_HEAD@GP_EX7(SYM,STR1)
EX_TEXT@} STR1;
FN_HEAD@GP_EX8(SYM)
EX_TEXT@\
FN_HEAD@GP_EX9(SYM)
EX_TEXT@\nSYM
FN_HEAD@GP_EY1(SYM)
EX_TEXT@}
FN_HEAD@GP_EY2(SYM)
EX_TEXT@SYM;
```

Notice that in some cases, the keywords are translated into the CPP macro substitution definition to be processed by the CPP before being compiled. The CPP processes it automatically through the inclusion of a CPP header file included using the specification language keyword RUNTIME_FILE. Note also that the translation done also involves concatenation which is not possible when using the CPP.

As the generic package e.g. **genpkg_STACK** is only to be used when needed, possibly taken from a libraries of packages, it is not required to be processed through the generic package preprocessor and as such it is implemented through the use of the CPP. Given next is the CPP header file **gen_pkg.h** which is used by the system to help implement the preprocessor.

```
#include <stdio.h>
#include "gentype.h"
#define GENERIC
#define CONST(x,y)
#define TYPE(x)
#define FN(x,y)
#define G_PACKAGE(x)
#define VIS  struct {
```

```
#define PROC(x)  void (*x)()
#define FUNC(x,y) x (*y)()
#define END_VIS(x)  } x;
#define PRIV  static struct {int iflg;
#define END_PRIV(x)  } private;
#define END_G_PACKAGE(x)
#define PRIV_INIT static void priv_init() { private.iflg=TRUE;
#define END }
#define EXPPKG(x) void LINK() {                    if (private.iflg!=TRUE) priv_init();
#define EXP(x,y)  x.y=y
#define END_EXPPKG  }
```

To demonstrate the usefulness of the newly acquired generic package facility, given below is all that is required to instantiate the package STACK to provide for elements of type REC.

```
#include "rec.h"
PKGINST(STACK,RSTACK)
INST(STACK_SIZE,100)
INST(STACK_TYPE,REC)
PKGLINK(getRSTACK)
END_PKGINST(STACK)
```

This file **inst_RSTACK.h** is the instantiation file while the calling program **Rmain.c** is given next :

```
#include "rec.h"
#include <string.h>

/*PPGPKG

GETPKG(RSTACK)
FUNC(struct rec,pop);
FUNC(int,push);
FUNC(bool,empty_stack);
FUNC(bool,full_stack);
struct rec top_of_stack;
END_GETPKG(RSTACK) */

main()
{
```

```
FILE *ifp=fopen("INFILE","r");
FILE *ofp=fopen("OUTFILE","w");
struct rec arec1,arec2;

/*PPGPKG getRSTACK();*/
while (!(feof(ifp)))
 { fscanf(ifp,"%s %d",arec1.name,&arec1.age);
   (*RSTACK.push)(arec1);
  if (strcmp(RSTACK.top_of_stack.name,"*")==0 || (*RSTACK.full_stack)())
   {arec2=(*RSTACK.pop)();
    while (((*RSTACK.empty_stack)())==FALSE)
      {arec2=(*RSTACK.pop)();
       fprintf(ofp,"%-10s %2d\n",arec2.name,arec2.age);
      }
   }
 }
}
```

5.5 : Highlights and Summary.

A technique has been presented here that introduces the concept of generic packages and abstract data types into C through careful and intelligent exploitation of the expansion,insertion and substitution facilities of the system and the CPP; and the characteristics of structures in C.

With a generic package facility, we can have a library of packages that can be instantiated and used immediately. As components of a package are implemented as components of a C structure, every visible components of the package can only be referred to with specific reference to the a newly instantiation package. Instantiations of two or more packages from one generic package will not pose any problem nor will any name clashes occur.

CHAPTER SIX

LOOKING AHEAD : FURTHER RESEARCH SUGGESTIONS AND POSSIBILITIES

We now come to the final chapter of this thesis, a chapter which discusses further areas of research that might be explored.

This thesis has demonstrated how a system can build preprocessors which implements C language extensions through specifying the specification of the preprocessor using a specification language. Thus, we can conclude that the system met the goals that were set up when the research work began, despite some limitations to the nature of the input programs, as already discussed in Chapter 4.

We now discuss possible modifications and extensions to the system. One possible extension would be to provide the user with a specification consistency check that will check whether the specifications specified by the user are in the correct format. As the specification file is processed using the language Awk, it is vital that the syntax and format of the language is strictly followed. In the present experimental system, a single blank line or a missing specification delimiter would invalidate an

entire specification. The check could be implemented in the form of a scanner that scans the specification file to detect irregularity in syntax and format and reports immediately the error in specification which might include a blank line, a missing delimiter, an incomplete specification or unbalanced curly brackets. Upon encountering a specification keyword, the format for specification of a particular keyword should be checked for conformity. Such checking would allow any error in specification to be detected at the earliest possible stage. However, up till now, no error in specification has managed to pass through all the stages leading to the building of the eventual preprocessor, without being detected.

Further modification could also be made to the interface built for specifying specifications to the system. At the moment, it is a simple interface which prompts the user with questions requiring a yes/no answer followed by prompts requesting input of a particular specification. A more elaborate interface would preferably be more informative in that it should be possible for the user to request the display of any production/production-rule and also give specifications with reference to the item being displayed.

An exciting further research project involving extending the system will be to modify the system to accept a set of language productions representing the grammar of a language together with the set of lexical entities to produce preprocessors for that language. This is perhaps a

generalisation of the work done described in this thesis. Thus, the modified system will initially read in the grammar productions from a grammar production file and the lexical entities from another or the same file and build syntax and lexical files **ppyffile** and **pplffile** respectively. Basically, to implement the proposed system we would require a front-end to our existing system that will process the grammar productions and lexical entities of the language into **ppyffile** and **pplffile**. This can be achieved by introducing the productions and lexical entities of the language in a format and specification readable by the front-end. The end product will be a preprocessor builder capable of building preprocessors for any language, given its grammar productions and lexical entities.

A step further would be to explore, with the help of the modified system, the field of language to language translation. With the use of the expansion and insertion facility of the system, simple language to language translation will no longer be a difficult task.

Another interesting research project would be to create a specification based system that will include a complete automation of the pre-post condition system and module specification interface with automatically generated checking routines based on specifications given by the user of the pre-post conditions, data-type invariants and module interfaces.

APPENDIX I : THE SPECIFICATION LANGUAGE PSL

```
"CNOTATION@"<notation>

"KEYWORD"
"@"<keyword>
<keyword>*
"@"

"PREFIX@"<string1>"@WITH@"<string2>

"NON_TERMINAL"
"@"<non-terminal>
<non-terminal>*
"@"

"INS_ACTION"
{"PROD@"<production_no>
{"RULE@"<production_rule_no>
{"ACT@"<symbol_no>
"@{"<action_line>+"}"
"@"}+}+}+

"INS_PRODRULE"
{"PROD@"<production_no>
{"PRULE"
"@"<production_rule>
"@"
{"ACT@"<symbol_no>
"@{"<action_line>+"}"
"@"}*}+}+

"ADD_NEWPROD"
{"LHS@"<non_terminal>
{"PRULE"
"@"<production_rule>
"@"
{"ACT@"<symbol_no>
"@{"<action_line>+"}"
"@"}*}+}+

"SYNT_INCFILE@"<filename>{","<filename>}*

"RUNTIME_FILE@"<filename>

"TEXT_EXP"
{"FN_HEAD@"<template_head>
"EX_TEXT@"<expansion_template>}+
```

Note : + denotes one or more. * denotes none or more. { and } groups items together.

APPENDIX II

CPBS C LANGUAGE PRODUCTIONS

```
[1] primary_expr
        : identifier
        | CONSTANT
        | STRING_LITERAL
        | '(' expr ')'
        ;

[2] postfix_expr
        : primary_expr
        | postfix_expr '[' expr ']'
        | postfix_expr '(' ')'
        | postfix_expr '(' argument_expr_list ')'
        | postfix_expr '.' identifier
        | postfix_expr PTR_OP identifier
        | postfix_expr INC_OP
        | postfix_expr DEC_OP
        ;

[3] argument_expr_list
        : assignment_expr
        | argument_expr_list ',' assignment_expr
        ;

[4] unary_expr
        : postfix_expr
        | INC_OP unary_expr
        | DEC_OP unary_expr
        | unary_operator cast_expr
        | SIZEOF unary_expr
        | SIZEOF '(' type_name ')'
        ;

[5] unary_operator
        : '&'
        | '*'
        | '+'
        | '-'
        | '~'
        | '!'
        ;

[6] cast_expr
        : unary_expr
        | '(' type_name ')' cast_expr
        ;

[7] multiplicative_expr
        : cast_expr
        | multiplicative_expr '*' cast_expr
```

```
        | multiplicative_expr '/' cast_expr
        | multiplicative_expr '%' cast_expr
        ;

[8] additive_expr
        : multiplicative_expr
        | additive_expr '+' multiplicative_expr
        | additive_expr '-' multiplicative_expr
        ;

[9] shift_expr
        : additive_expr
        | shift_expr LEFT_OP additive_expr
        | shift_expr RIGHT_OP additive_expr
        ;

[10] relational_expr
        : shift_expr
        | relational_expr '<' shift_expr
        | relational_expr '>' shift_expr
        | relational_expr LE_OP shift_expr
        | relational_expr GE_OP shift_expr
        ;

[11] equality_expr
        : relational_expr
        | equality_expr EQ_OP relational_expr
        | equality_expr NE_OP relational_expr
        ;

[12] and_expr
        : equality_expr
        | and_expr '&' equality_expr
        ;

[13] exclusive_or_expr
        : and_expr
        | exclusive_or_expr '^' and_expr
        ;

[14] inclusive_or_expr
        : exclusive_or_expr
        | inclusive_or_expr '|' exclusive_or_expr
        ;

[15] logical_and_expr
        : inclusive_or_expr
        | logical_and_expr AND_OP inclusive_or_expr
        ;

[16] logical_or_expr
        : logical_and_expr
        | logical_or_expr OR_OP logical_and_expr
        ;
```

```
[17] conditional_expr
        : logical_or_expr
        | logical_or_expr '?' logical_or_expr ':' conditional_expr
        ;

[18] assignment_expr
        : conditional_expr
        | unary_expr assignment_operator assignment_expr
        ;

[19] assignment_operator
        : '='
        | MUL_ASSIGN
        | DIV_ASSIGN
        | MOD_ASSIGN
        | ADD_ASSIGN
        | SUB_ASSIGN
        | LEFT_ASSIGN
        | RIGHT_ASSIGN
        | AND_ASSIGN
        | XOR_ASSIGN
        | OR_ASSIGN
        ;

[20] expr
        : assignment_expr
        | expr ',' assignment_expr
        ;

[21] constant_expr
        : conditional_expr
        ;

[22] declaration
        : declaration_specifiers ';'
        | declaration_specifiers init_declarator_list ';'
        | COM declaration_specifiers ';'
        | COM declaration_specifiers init_declarator_list ';'
        ;

[23] declaration_specifiers
        : storage_class_specifier
        | storage_class_specifier declaration_specifiers
        | type_specifier
        | type_specifier declaration_specifiers
        ;

[24] init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator
        ;

[25] init_declarator
        : declarator
```

```
        | declarator '=' initializer
        ;

[26] storage_class_specifier
        : TYPEDEF
        | EXTERN
        | STATIC
        | AUTO
        | REGISTER
        ;

[27] type_specifier
        : CHAR
        | SHORT
        | INT
        | LONG
        | SIGNED
        | UNSIGNED
        | FLOAT
        | DOUBLE
        | CONST
        | VOLATILE
        | VOID
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        | FILE
        ;

[28] struct_or_union_specifier
        : struct_or_union identifier '{' struct_declaration_list '}'
        | struct_or_union '{' struct_declaration_list '}'
        | struct_or_union identifier
        ;

[29] struct_or_union
        : STRUCT
        | UNION
        ;

[30] struct_declaration_list
        : struct_declaration
        | struct_declaration_list struct_declaration
        ;

[31] struct_declaration
        : type_specifier_list struct_declarator_list ';'
        ;

[32] struct_declarator_list
        : struct_declarator
        | struct_declarator_list ',' struct_declarator
        ;
```

```
[33] struct_declarator
        : declarator
        | ':' constant_expr
        | declarator ':' constant_expr
        ;

[34] enum_specifier
        : ENUM '{' enumerator_list '}'
        | ENUM identifier '{' enumerator_list '}'
        | ENUM identifier
        ;

[35] enumerator_list
        : enumerator
        | enumerator_list ',' enumerator
        ;

[36] enumerator
        : identifier
        | identifier '=' constant_expr
        ;

[37] declarator
        : declarator2
        | pointer declarator2
        | COM declarator2
        | COM pointer declarator2
        ;

[38] declarator2
        : identifier
        | '(' declarator ')'
        | declarator2 '[' ']'
        | declarator2 '[' constant_expr ']'
        | declarator2 '(' ')'
        | declarator2 '(' parameter_type_list ')'
        | declarator2 '(' parameter_identifier_list ')'
        ;

[39] pointer
        : '*'
        | '*' type_specifier_list
        | '*' pointer
        | '*' type_specifier_list pointer
        ;

[40] type_specifier_list
        : type_specifier
        | type_specifier_list type_specifier
        ;

[41] parameter_identifier_list
        : identifier_list
        | identifier_list ',' ELIPSIS
```

```
        ;

[42] identifier_list
        : identifier
        | identifier_list ',' identifier
        ;

[43] parameter_type_list
        : parameter_list
        | parameter_list ',' ELIPSIS
        ;

[44] parameter_list
        : parameter_declaration
        | parameter_list ',' parameter_declaration
        ;

[45] parameter_declaration
        : type_specifier_list declarator
        | type_name
        ;

[46] type_name
        : type_specifier_list
        | type_specifier_list abstract_declarator
        ;

[47] abstract_declarator
        : pointer
        | abstract_declarator2
        | pointer abstract_declarator2
        ;

[48] abstract_declarator2
        : '(' abstract_declarator ')'
        | '[' ']'
        | '[' constant_expr ']'
        | abstract_declarator2 '[' ']'
        | abstract_declarator2 '[' constant_expr ']'
        | '(' ')'
        | '(' parameter_type_list ')'
        | abstract_declarator2 '(' ')'
        | abstract_declarator2 '(' parameter_type_list ')'
        ;

[49] initializer
        : assignment_expr
        | '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        ;

[50] initializer_list
        : initializer
        | initializer_list ',' initializer
```

```
        ;

[51] statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        | prepro_statement
        ;

[52] prepro_statement
        : COM
;

[53] labeled_statement
        : identifier ':' statement
        | CASE constant_expr ':' statement
        | DEFAULT ':' statement
        ;

[54] compound_statement
        : begin end
        | begin statement_list end
        | begin declaration_list end
        | begin declaration_list statement_list end
        ;

[55] begin
        : '{'
        | '{' COM
        ;

[56] end
        : '}'
        ;

[57] declaration_list
        : declaration
        | declaration_list declaration
        ;

[58] statement_list
        : statement
        | statement_list statement
        ;

[59] expression_statement
        : ';'
        | expr ';'
        ;

[60] selection_statement
```

```
        : IF '(' expr ')' statement
        | IF '(' expr ')' statement ELSE statement
        | SWITCH '(' expr ')' statement
        ;

[61] iteration_statement
        : WHILE '(' expr ')' statement
        | DO statement WHILE '(' expr ')' ';'
        | FOR '(' ';' ';' ')' statement
        | FOR '(' ';' ';' expr ')' statement
        | FOR '(' ';' expr ';' ')' statement
        | FOR '(' ';' expr ';' expr ')' statement
        | FOR '(' expr ';' ';' ')' statement
        | FOR '(' expr ';' ';' expr ')' statement
        | FOR '(' expr ';' expr ';' ')' statement
        | FOR '(' expr ';' expr ';' expr ')' statement
        ;

[62] jump_statement
        : GOTO identifier ';'
        | CONTINUE ';'
        | BREAK ';'
        | RETURN ';'
        | RETURN expr ';'
        ;

[63] filestart
        : file
        ;

[64] file
        : external_definition
        | file external_definition
        ;

[65] external_definition
        : function_definition
        | declaration
        | com
        ;

[66] com
        : COM
        | com COM
        ;

[67] function_definition
        : declarator function_body
        | declaration_specifiers declarator function_body
        ;

[68] function_body
        : compound_statement
        | declaration_list compound_statement
```

```
        ;

[69] identifier
        : IDENTIFIER
        ;
```

APPENDIX III : CPBS Samples of Generated Files

File : Sample of Lex generated preprocessing file, genpkg.lf (given below)

```
D                       [0-9]
L                       [a-zA-Z_]
H                       [a-fA-F0-9]
E                       [Ee][+-]?{D}+
FS                      (f|F|l|L)
IS                      (u|U|l|L)*
%p 5000
%{
#include <stdio.h>
#include <string.h>
#include "basic.h"
#include "y.tab.h"
FILE *fopen(),*fclose();
FILE *ppout_file;
char *ppbuffer;
int ppcomcount=0,pplncount,ppchcount;
void assign(),assign2(),count(),flcount(),prefix();
%}
%%
"/*""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"  {
                if (strncmp(yytext,"/*PP",4)==0) REJECT; }
"/*PP""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/"  {
if (strncmp(yytext,"/*PPGPKG",8)!=0)
{
if ((++ppcomcount)==1)
ppout_file=fopen("ppcomfile","w");
ppbuffer=yytext;
pplncount=0;ppchcount=0;
while (yytext[ppchcount]!='\0')
if (yytext[ppchcount++]=='\n')  ++pplncount;
fprintf(ppout_file,"%d ",++pplncount);
fputs(ppbuffer,ppout_file);
fprintf(ppout_file,"\n");
strcpy(yytext,"/*COM*/");
assign();count();return(COM);
}
else REJECT;
}
"/*PPGPKG""/"*([^*/]|[^*]"/"|"*"[^/])*"*"*"*/" {yyless(8);}
"PKGINST" {assign();return(PKGINST);}
"INST" {assign();return(INST);}
"PKGLINK" {assign();return(PKGLINK);}
"END_PKGINST" {assign();return(END_PKGINST);}
"GETPKG" {assign();return(GETPKG);}
"END_GETPKG" {assign();return(END_GETPKG);}
"FUNC" {assign();return(FUNC);}
"PROC" {assign();return(PROC);}
```

```
"*/"            {;}
"auto"                  { assign();count(); return(AUTO); }
"break"                 { assign();count(); return(BREAK); }
"case"                  { assign();count(); return(CASE); }
"char"                  { assign();count(); return(CHAR); }
"const"                 { assign();count(); return(CONST); }
"continue"              { assign();count(); return(CONTINUE); }
"default"               { assign();count(); return(DEFAULT); }
"do"                    { assign();count(); return(DO); }
"double"                { assign();count(); return(DOUBLE); }
"else"                  { assign();count(); return(ELSE); }
"enum"                  { assign();count(); return(ENUM); }
"extern"                { assign();count(); return(EXTERN); }
"float"                 { assign();count(); return(FLOAT); }
"for"                   { assign();count(); return(FOR); }
"goto"                  { assign();count(); return(GOTO); }
"if"                    { assign();count(); return(IF); }
"int"                   { assign();count(); return(INT); }
"long"                  { assign();count(); return(LONG); }
"register"              { assign();count(); return(REGISTER); }
"return"                { assign();count(); return(RETURN); }
"short"                 { assign();count(); return(SHORT); }
"signed"                { assign();count(); return(SIGNED); }
"sizeof"        { assign();count(); return(SIZEOF); }
"static"        { assign();count(); return(STATIC); }
"struct"        { assign();count(); return(STRUCT); }
"switch"                { assign();count(); return(SWITCH); }
"typedef"               { assign();count(); return(TYPEDEF); }
"FILE"                  { assign();count(); return(FTOKEN); }
"union"                 { assign();count(); return(UNION); }
"unsigned"              { assign();count(); return(UNSIGNED); }
"void"                  { assign();count(); return(VOID); }
"volatile"              { assign();count(); return(VOLATILE); }
"while"                 { assign();count(); return(WHILE); }
{L}({L}|{D})*           { assign();count(); return(check_type()); }
0[xX]{H}+{IS}?          { assign();count(); return(CONSTANT); }
0[xX]{H}+{IS}?          { assign();count(); return(CONSTANT); }
0{D}+{IS}?              { assign();count(); return(CONSTANT); }
0{D}+{IS}?              { assign();count(); return(CONSTANT); }
{D}+{IS}?               { assign();count(); return(CONSTANT); }
{D}+{IS}?               { assign();count(); return(CONSTANT); }
{D}+{E}{FS}?            { assign();count(); return(CONSTANT); }
{D}*"."{D}+({E})?{FS}? { assign();count(); return(CONSTANT); }
{D}+"."{D}*({E})?{FS}? { assign();count(); return(CONSTANT); }

">>="                   { assign();count(); return(RIGHT_ASSIGN); }
"<<="                   { assign();count(); return(LEFT_ASSIGN); }
"+="                    { assign();count(); return(ADD_ASSIGN); }
"-="                    { assign();count(); return(SUB_ASSIGN); }
"*="                    { assign();count(); return(MUL_ASSIGN); }
"/="                    { assign();count(); return(DIV_ASSIGN); }
"%="                    { assign();count(); return(MOD_ASSIGN); }
"&="                    { assign();count(); return(AND_ASSIGN); }
"^="                    { assign();count(); return(XOR_ASSIGN); }
```

```
"|="                    { assign();count(); return(OR_ASSIGN); }
">>"                    { assign();count(); return(RIGHT_OP); }
"<<"                    { assign();count(); return(LEFT_OP); }
"++"                    { assign();count(); return(INC_OP); }
"--"                    { assign();count(); return(DEC_OP); }
"->"                    { assign();count(); return(PTR_OP); }
"&&"                    { assign();count(); return(AND_OP); }
"||"                    { assign();count(); return(OR_OP); }
"<="                    { assign();count(); return(LE_OP); }
">="                    { assign();count(); return(GE_OP); }
"=="                    { assign();count(); return(EQ_OP); }
"!="                    { assign();count(); return(NE_OP); }
";"                     { assign();count(); return(SC); }
"{"                     { assign();count(); return(LC); }
"}"                     { assign();count(); return(RC); }
","                     { assign();count(); return(CM); }
":"                     { assign();count(); return(CL); }
"="                     { assign();count(); return(EQ); }
"("                     { assign();count(); return(LP); }
")"                     { assign();count(); return(RP); }
"["                     { assign();count(); return(LB); }
"]"                     { assign();count(); return(RB); }
"."                     { assign();count(); return(DT); }
"&"                     { assign();count(); return(AP); }
"!"                     { assign();count(); return(NT); }
"~"                     { assign();count(); return(TL); }
"-"                     { assign();count(); return(MN); }
"+"                     { assign();count(); return(PL); }
"*"                     { assign();count(); return(AS); }
"/"                     { assign();count(); return(DV); }
"%"                     { assign();count(); return(PC); }
"<"                     { assign();count(); return(LT); }
">"                     { assign();count(); return(GT); }
"^"                     { assign();count(); return(EX); }
"|"                     { assign();count(); return(OR); }
"?"                     { assign();count(); return(QS); }
"\n"            { flcount();}
[ \t\v\n\f]             { count(); }
%%
#include "lstd.h"
```

File : Sample of generated Yacc preprocessing file, genpkg.yf (given below)

```
%{
#include "basic.h"
%}
%union { int val;
        Symbol *sym;
      }
%token <sym> AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN
%tolen <sym> ADD_ASSIGN
%token <sym> CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO
```

```
%token <sym> CONTINUE BREAK RETURN
%token <sym> CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE
%token <sym> CONST VOLATILE VOID
%token <sym> END_GETPKG
%token <sym> END_PKGINST
%token <sym> FTOKEN COM IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token <sym> FUNC
%token <sym> GETPKG
%token <sym> INST
%token <sym> LP RP LC RC LB RB EQ LT GT PL MN AS DV PC QS DT OR AP
%token <sym> NT SC CL CM TL EX
%token <sym> PKGINST
%token <sym> PKGLINK
%token <sym> PROC
%token <sym> PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP
%token <sym> EQ_OP NE_OP
%token <sym> STRUCT UNION ENUM ELIPSIS GTNGE
%token <sym> SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token <sym> TYPEDEF EXTERN STATIC AUTO REGISTER
%token <sym> XOR_ASSIGN OR_ASSIGN TYPE_NAME
%type <sym> abstract_declarator abstract_declarator2 initializer initializer_list
%type <sym> any_type
%type <sym> assignment_expr assignment_operator expr constant_expr declaration
%type <sym> begin end cast_expr multiplicative_expr additive_expr shift_expr
%type <sym> declaration_specifiers init_declarator_list init_declarator
%type <sym> enumerator_list enumerator declarator declarator2 pointer
%type <sym> filestart primary_expr postfix_expr argument_expr_list unary_expr
%type <sym> unary_operator
%type <sym> function_definition function_body identifier prepro_statement com
%type <sym> inclusive_or_expr logical_and_expr logical_or_expr conditional_expr
%type <sym> instpkg
%type <sym> iteration_statement jump_statement file external_definition
%type <sym> mainprog
%type <sym> num_ident
%type <sym> parameter_type_list parameter_list parameter_declaration type_name
%type <sym> relational_expr equality_expr and_expr exclusive_or_expr
%type <sym> statement labeled_statement compound_statement declaration_list
%type <sym> statement_list expression_statement selection_statement
%type <sym> storage_class_specifier type_specifier struct_or_union_specifier
%type <sym> struct_declarator_list struct_declarator enum_specifier
%type <sym> struct_or_union struct_declaration_list struct_declaration
%type <sym> type_specifier_list parameter_identifier_list identifier_list
%start filestart
%%
primary_expr
 : identifier
 | CONSTANT
 | STRING_LITERAL
 | LP
 expr
RP {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
```

```
postfix_expr
 : primary_expr
 | postfix_expr
 LB
 expr
RB {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
 | postfix_expr
 LP
RP {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | postfix_expr
 LP
 argument_expr_list
RP {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
 | postfix_expr
 DT
identifier {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | postfix_expr
 PTR_OP
identifier {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | postfix_expr
INC_OP {
 $$=ppjoin($1,$2);
}
 | postfix_expr
DEC_OP {
 $$=ppjoin($1,$2);
}
;
argument_expr_list
 : assignment_expr
 | argument_expr_list
 CM
assignment_expr
{
 $$=ppjoin($1,ppjoin($2,$3));
}
;
unary_expr
 : postfix_expr
 | INC_OP
unary_expr {
 $$=ppjoin($1,$2);
 }
 | DEC_OP
```

```
unary_expr {
 $$=ppjoin($1,$2);
 }
 | unary_operator
cast_expr {
 $$=ppjoin($1,$2);
 }
 | SIZEOF
unary_expr {
 $$=ppjoin($1,$2);
}
 | SIZEOF
 LP
 type_name
RP {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
 }
;
unary_operator
 : AP
 | AS
 | PL
 | MN
 | TL
 | NT
;
cast_expr
 : unary_expr
 | LP
 type_name
 RP
cast_expr {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
;
multiplicative_expr
 : cast_expr
 | multiplicative_expr
 AS
cast_expr {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | multiplicative_expr
 DV
cast_expr {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | multiplicative_expr
 PC
cast_expr {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
additive_expr
```

```
 : multiplicative_expr
 | additive_expr
 PL
multiplicative_expr {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | additive_expr
 MN
multiplicative_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
shift_expr
 : additive_expr
 | shift_expr
 LEFT_OP
additive_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
 | shift_expr
 RIGHT_OP
additive_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
relational_expr
 : shift_expr
 | relational_expr
 LT
shift_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
 | relational_expr
 GT
shift_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
 | relational_expr
 LE_OP
shift_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
 | relational_expr
 GE_OP
shift_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
equality_expr
 : relational_expr
 | equality_expr
 EQ_OP
relational_expr {
 $$=ppjoin($1,ppjoin($2,$3));
```

```
 }
 | equality_expr
 NE_OP
relational_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
and_expr
 : equality_expr
 | and_expr
 AP
equality_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
exclusive_or_expr
 : and_expr
 | exclusive_or_expr
 EX
and_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
inclusive_or_expr
 : exclusive_or_expr
 | inclusive_or_expr
 OR
exclusive_or_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
logical_and_expr
 : inclusive_or_expr
 | logical_and_expr
 AND_OP
inclusive_or_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
logical_or_expr
 : logical_and_expr
 | logical_or_expr
 OR_OP
logical_and_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
conditional_expr
 : logical_or_expr
 | logical_or_expr
 QS
 logical_or_expr
 CL
conditional_expr {
```

```
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,$5))));
 }
;
assignment_expr
 : conditional_expr
 | unary_expr
 assignment_operator
assignment_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
assignment_operator
 : EQ
 | MUL_ASSIGN
 | DIV_ASSIGN
 | MOD_ASSIGN
 | ADD_ASSIGN
 | SUB_ASSIGN
 | LEFT_ASSIGN
 | RIGHT_ASSIGN
 | AND_ASSIGN
 | XOR_ASSIGN
 | OR_ASSIGN
;
expr
 : assignment_expr
 | expr
 CM
assignment_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
constant_expr
 : conditional_expr
;
declaration
 : declaration_specifiers
SC {
 $$=ppjoin($1,$2);
}
 | declaration_specifiers
 init_declarator_list
SC {
 $$=ppjoin($1,ppjoin($2,$3));
 }
 | COM
 declaration_specifiers
SC {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | COM
 declaration_specifiers
 init_declarator_list
SC {
```

```
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
 }
;
declaration_specifiers
 : storage_class_specifier
 | storage_class_specifier
declaration_specifiers {
 $$=ppjoin($1,$2);
 }
 | type_specifier
 | type_specifier
declaration_specifiers {
 $$=ppjoin($1,$2);
 }
;
init_declarator_list
 : init_declarator
 | init_declarator_list
 CM
init_declarator {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
init_declarator
 : declarator
 | declarator
 EQ
initializer {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
storage_class_specifier
 : TYPEDEF
 | EXTERN
 | STATIC
 | AUTO
 | REGISTER
;
type_specifier
 : CHAR
 | SHORT
 | INT
 | LONG
 | SIGNED
 | UNSIGNED
 | FLOAT
 | DOUBLE
 | CONST
 | VOLATILE
 | VOID
 | struct_or_union_specifier
 | enum_specifier
 | TYPE_NAME
 | FTOKEN
```

```
;
struct_or_union_specifier
 : struct_or_union
 identifier
 LC
 struct_declaration_list
RC {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,$5))));
 }
 | struct_or_union
 LC
 struct_declaration_list
RC {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
 }
 | struct_or_union
identifier {
 $$=ppjoin($1,$2);
 }
;
struct_or_union
 : STRUCT
 | UNION
;
struct_declaration_list
 : struct_declaration
 | struct_declaration_list
struct_declaration {
 $$=ppjoin($1,$2);
 }
;
struct_declaration
 : type_specifier_list
 struct_declarator_list
SC {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
struct_declarator_list
 : struct_declarator
 | struct_declarator_list
 CM
struct_declarator {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
struct_declarator
 : declarator
 | CL
constant_expr {
 $$=ppjoin($1,$2);
 }
 | declarator
 CL
```

```
constant_expr {
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
enum_specifier
 : ENUM
 LC
 enumerator_list
RC {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
 }
 | ENUM
 identifier
 LC
 enumerator_list
RC {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,$5))));
 }
 | ENUM
identifier {
 $$=ppjoin($1,$2);
}
;
enumerator_list
 : enumerator
 | enumerator_list
 CM
enumerator {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
enumerator
 : identifier
 | identifier
 EQ
constant_expr {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
declarator
 : declarator2
 | pointer
declarator2 {
 $$=ppjoin($1,$2);
}
 | COM
declarator2 {
 $$=ppjoin($1,$2);
 }
 | COM
 pointer
declarator2 {
 $$=ppjoin($1,ppjoin($2,$3));
}
```

```
;
declarator2
 : identifier
 | LP
 declarator
RP {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | declarator2
 LB
RB {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | declarator2
 LB
 constant_expr
RB {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
 | declarator2
 LP
RP {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | declarator2
 LP
 parameter_type_list
RP {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
 | declarator2
 LP
 parameter_identifier_list
RP {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
;
pointer
 : AS
 | AS
type_specifier_list {
 $$=ppjoin($1,$2);
}
 | AS
pointer {
 $$=ppjoin($1,$2);
}
 | AS
 type_specifier_list
pointer {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
type_specifier_list
```

```
 : type_specifier
 | type_specifier_list
type_specifier {
 $$=ppjoin($1,$2);
}
;
parameter_identifier_list
 : identifier_list
 | identifier_list
 CM
ELIPSIS {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
identifier_list
 : identifier
 | identifier_list
 CM
identifier {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
parameter_type_list
 : parameter_list
 | parameter_list
 CM
ELIPSIS {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
parameter_list
 : parameter_declaration
 | parameter_list
 CM
parameter_declaration {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
parameter_declaration
 : type_specifier_list
declarator {
 $$=ppjoin($1,$2);
}
 | type_name
;
type_name
 : type_specifier_list
 | type_specifier_list
abstract_declarator {
 $$=ppjoin($1,$2);
 }
;
abstract_declarator
 : pointer
```

```
| abstract_declarator2
| pointer
abstract_declarator2 {
 $$=ppjoin($1,$2);
}
;
abstract_declarator2
 : LP
 abstract_declarator
 RP {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | LB
RB {
 $$=ppjoin($1,$2);
}
 | LB
 constant_expr
RB {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | abstract_declarator2
 LB
RB {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | abstract_declarator2
 LB
 constant_expr
RB {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
 | LP
RP {
 $$=ppjoin($1,$2);
}
 | LP
 parameter_type_list
RP {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | abstract_declarator2
 LP
RP {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | abstract_declarator2
 LP
 parameter_type_list
RP {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
;
initializer
```

```
: assignment_expr
| LC
 initializer_list
RC {
 $$=ppjoin($1,ppjoin($2,$3));
}
| LC
 initializer_list
 CM
RC {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
;
initializer_list
 : initializer
 | initializer_list
 CM
initializer {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
statement
 : labeled_statement
 | compound_statement
 | expression_statement
 | selection_statement
 | iteration_statement
 | jump_statement
 | prepro_statement
;
prepro_statement
 : COM
;
labeled_statement
 : identifier
 CL
statement {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | CASE
 constant_expr
 CL
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
 | DEFAULT
 CL
statement {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
compound_statement
 : begin
end {
```

```
 $$=ppjoin($1,$2);
}
 | begin
 statement_list
end {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | begin
 declaration_list
end {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | begin
 declaration_list
 statement_list
end {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,$4)));
}
;
begin
 : LC
 | LC
COM {
 $$=ppjoin($1,$2);
 }
;
end
 : RC
;
declaration_list
 : declaration
 | declaration_list
declaration {
 $$=ppjoin($1,$2);
}
;
statement_list
 : statement
 | statement_list
statement {
 $$=ppjoin($1,$2);
}
;
expression_statement
 : SC
 | expr
SC {
 $$=ppjoin($1,$2);
}
;
selection_statement
 : IF
 LP
 expr
```

```
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,$5))));
}
 | IF
 LP
 expr
 RP
 statement
 ELSE
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,$7))))));
}
 | SWITCH
 LP
 expr
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,$5))));
}
;
iteration_statement
 : WHILE
 LP
 expr
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,$5))));
}
 | DO
 statement
 WHILE
 LP
 expr
 RP
SC {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,$7))))));
}
 | FOR
 LP
 SC
 SC
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,$6)))));
}
 | FOR
 LP
 SC
 SC
 expr
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,$7))))));
```

```
}
 | FOR
 LP
 SC
 expr
 SC
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,$7))))));
}
 | FOR
 LP
 SC
 expr
 SC
 expr
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,ppjoin($7,$8)))))));
}
 | FOR
 LP
 expr
 SC
 SC
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,$7))))));
}
 | FOR
 LP
 expr
 SC
 SC
 expr
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,ppjoin($7,$8)))))));
}
 | FOR
 LP
 expr
 SC
 expr
 SC
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,ppjoin($7,$8)))))));
}
 | FOR
 LP
 expr
 SC
 expr
```

```
 SC
 expr
 RP
statement {
 $$=ppjoin($1,ppjoin($2,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,ppjoin($7,ppjoin($8,$9)))))));
}
;
jump_statement
 : GOTO
 identifier
SC {
 $$=ppjoin($1,ppjoin($2,$3));
}
 | CONTINUE
SC {
 $$=ppjoin($1,$2);
}
 | BREAK
SC {
 $$=ppjoin($1,$2);
}
 | RETURN
SC {
 $$=ppjoin($1,$2);
}
 | RETURN
 expr
SC {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
filestart
:file {
fclose(out_file);
printf("\n#include \"gen_pkg.h\" \n");
pprlist($1);printf("\n");
}
;
file
 : file
external_definition {
 $$=ppjoin($1,$2);
}
 | external_definition
;
external_definition
 : function_definition
 | declaration
 | com
 | instpkg
 | mainprog
 | any_type
 identifier
 SC {
```

```
 $$=ppjoin($1,ppjoin($2,$3));
 }
;
com
 : COM
 | com
 COM {
 $$=ppjoin($1,$2);
 }
;
function_definition
 : declarator
function_body {
 $$=ppjoin($1,$2);
}
 | declaration_specifiers
 declarator
function_body {
 $$=ppjoin($1,ppjoin($2,$3));
}
;
function_body
 : compound_statement
 | declaration_list
compound_statement {
 $$=ppjoin($1,$2);
}
;
identifier
 : IDENTIFIER
;
instpkg
: PKGINST {
{
{
ppstrptr=pplinkup($1);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"\n#define");
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$1=ppexpand(ppoutptr,pptextarr);
}
}
 }
 LP
 identifier
 CM
 identifier
```

```
 RP {
 $$=ppjoin($1,ppjoin($4,$6));
 }
| PKGLINK {
{
{
ppstrptr=pplinkup($1);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"\n#define LINK");
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$1=ppexpand(ppoutptr,pptextarr);
}
}
 }
 LP
 identifier
 RP {
 $$=ppjoin($1,$4);
 }
| INST {
{
{
ppstrptr=pplinkup($1);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"\n#define");
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$1=ppexpand(ppoutptr,pptextarr);
}
}
 }
 LP
 identifier
 CM
 num_ident
 RP {
 $$=ppjoin($1,ppjoin($4,$6));
 }
| END_PKGINST {
{
{
```

```
ppstrptr=pplinkup($1);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"\n#include");
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$1=ppexpand(ppoutptr,pptextarr);
}
}
 }
 LP
 identifier {
{
{
ppstrptr=pplinkup($4);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"\"genpkg_%s\"",pptestarr);
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[r++]=*(pptextptr++);
pptextarr[ppr]='\0';
$4=ppexpand(ppoutptr,pptextarr);
}
}
 }
 RP {
 $$=ppjoin($1,$4);
 }
;
mainprog
: GETPKG {
{
{
ppstrptr=pplinkup($1);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"\nextern");
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$1=ppexpand(ppoutptr,pptextarr);
```

```
}
}
 }
 LP
 identifier {
{
temp1=$4->name;
thisptr=anarr;
sprintf(thisptr,"%s",temp1);
{
ppstrptr=pplinkup($4);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"struct { \n");
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$4=ppexpand(ppoutptr,pptextarr);
}
}
 }
 RP {
 $$=ppjoin($1,$4);
 }
| END_GETPKG {
{
{
ppstrptr=pplinkup($1);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"}");
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$1=ppexpand(ppoutptr,pptextarr);
}
}
 }
 LP
 identifier {
{
{
ppstrptr=pplinkup($4);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
```

```
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"%s;\n",pptestarr);
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$4=ppexpand(ppoutptr,pptextarr);
}
}
 }
 RP {
 $$=ppjoin($1,$4);
 }
| FUNC {
{
{
ppstrptr=pplinkup($1);
ppb=0;
while ((*ppstrptr)!='\0')
pptestarr[ppb++]=*(ppstrptr++);
pptestarr[--ppb]='\0';
pptextptr=pptextarr;
sprintf(pptextptr,"\n%s",pptestarr);
ppr=0;
while ((*pptextptr)!='\0')
pptextarr[ppr++]=*(pptextptr++);
pptextarr[ppr]='\0';
$1=ppexpand(ppoutptr,pptextarr);
}
}
 }
 LP
 any_type
 CM
 identifier
 RP
 SC {
 $$=ppjoin($1,ppjoin($3,ppjoin($4,ppjoin($5,ppjoin($6,ppjoin($7,$8))))));
 }
;
num_ident
 : identifier
 | CONSTANT
 | type_specifier
;
any_type
 : type_specifier
 | identifier
;
%%
#include <stdio.h>
#include <string.h>
#include "ystd.h"
```

```
#include "gpkg.h"
```

APPENDIX IV : Preprocessors Variables and Routines Files

File : pre- and post-condition preprocessor : ppcdef.h (given below)

```
char *thisptr,*tstrptr,*temp1,*temp2;
char anarr[1000];
int checkflag=0,levcount=0;
```

File : index checker : sfxstruc.h (given below)

```
#define ARRNMSIZE 20
#define MAXDIM    10
#define LEVDIM    20

#define ENTRY struct entry

ENTRY
   { char arrname[ARRNMSIZE];
    int dimsize[MAXDIM+1];
    ENTRY *next,*prev;
   };
```

File : index checker : sfxstack.h (given below)

```
#define SITEM struct stackitem
SITEM
   {int val;
    SITEM *next;
   } *top;

void push(value) /* pushes value on top of stack */
 int value;
{SITEM *newptr;
newptr=(SITEM *)malloc(sizeof(SITEM));
newptr->val=value;
newptr->next=top;
top=newptr;
}

void pop() /* pops an item off the stack */
{SITEM *poptr;

poptr=top;
top=top->next;
poptr->next=(SITEM *)0;
}
```

File : index checker : sfxdef.h (given below)

```
ENTRY *evar_head,*evar_end,*fvar_head,*fvar_end,*ptr[LEVDIM],*temp_entry;
int levcount=0,fnflag=0,notproper=0,arrflag[20],dimcount=0,tc=(-1),ptrflag[20];
int k,t=0,closebrac=0;
int storeflag=0,checkflag=0,sigflag=0;

ENTRY *store(s1)  /* stores array names */
char *s1;

{ENTRY *pointr;

pointr=(ENTRY *)malloc(sizeof(ENTRY));
strcpy(&(pointr->arrname[0]),s1);
pointr->next=(ENTRY *)0;
pointr->prev=(ENTRY *)0;
return(pointr);
}

int thesize(astring,count)  /* retrieves the size of an array dimension */
 char astring[];
 int count;
 {
 ENTRY *q;

 q=fvar_end;
 while (q!=0 && strcmp(q->arrname,astring)!=0)
        q=q->prev;
 if (q==0)
        {
         q=evar_end;
         while (q!=0 && strcmp(q->arrname,astring)!=0)
          q=q->prev;
        }
 if (q==0) return(0);
   else
         return(q->dimsize[count]);
 }

 void delete(pointr) /* deletes pointer from fn var list */
 ENTRY *pointr;
 {ENTRY *q;
 q=pointr;
 if ((q->prev)==(ENTRY *)0)
     {fvar_head=(ENTRY *)0;
      fvar_end=(ENTRY *)0;
     }
 else
    {q=q->prev;
     q->next=(ENTRY *)0;
     fvar_end=q;
    }
}


void fnlink(pointr) /* links pointer to fn list */
```

```
ENTRY *pointr;
{ if (fvar_head==0)
    {fvar_head=pointr;
     fvar_end=pointr;
    }
  else
   {fvar_end->next=pointr;
    pointr->prev=fvar_end;
    fvar_end=pointr;
   }
}



void extlink(pointr) /* links pointer to external list */
ENTRY *pointr;
{ if (evar_head==0)
    {evar_head=pointr;
     evar_end=pointr;
    }
  else
   {evar_end->next=pointr;
    pointr->prev=evar_end;
    evar_end=pointr;
   }
}
void condfn()   /* conditional subroutine use */
{
 if ((arrflag[levcount]==1) && ((sigflag ==1) || (storeflag==1)))
      {
       if (fnflag==1)
         {
          fnlink(temp_entry);
          if (levcount>tc) {
                        ptr[levcount]=temp_entry;
                        ptrflag[levcount]=1;
                       }
                   tc=levcount;
          }
       else
       {extlink(temp_entry);
       }

     temp_entry=(ENTRY *)0;
     arrflag[levcount]=0;
     if (sigflag==1)  sigflag=0;
     }
}
```

File : generic package preprocessor : gpkg.h

```
char *thisptr,*temp1,anarr[1000];
```

APPENDIX V : CPBS USER MANUAL

An Overview

This manual describes how to use a C language C Preprocessor Building System (CPBS) under the Unix system. CPBS accepts a specification of a preprocessor written by the user in a specification file using a specification language and generates the required preprocessor. The specification language to be used is as defined in Appendix I of this thesis.

The specification includes additional codes for a symbol term in a production-rule, rules representing extended productions or additional productions together with codes to be invoked when the rules are recognised. References to the grammar of the C language is based on the set of modified C grammar productions listed in the Appendix II of this thesis.

The Specification Language

A specification file is formatted with instructive keywords and fields representing specific instructions to the system. It is used to state the requirements of the user on the types of functions that are required. The language used to specify the preprocessor is known as the Preprocessor Specifcation Language (PSL). A definition of PSL is in Appendix I of the thesis. References to an example made in the description of the keywords below refers to an example of a specification file given in the next section. The keywords of PSL are as follows :

CNOTATION :

This keyword refers to the comment notation to be recognised by the intended preprocessor as an instructive comment header to preprocessing instructions i.e. for example CNOTATION@COND instructs the system to build a preprocessor that will only process preprocessing instructions of the form /*PPCOND.....*/.

KEYWORD :

This keyword instructs the system to recognised instructive keywords of the intended preprocessor which are also to be used in the inserted production-rules. Keywords that are not to be printed to the output should be postfixed with '(NP)'.

INS_ACTION :

This keyword instructs the insertion of an action after a

particular symbol term i.e. to say that the user intends to execute an action after the symbol has been recognised during parsing in the context of the production-rule. Insertion of an action is with reference to the symbol term position. Action can be a set of C codes or a call to a routine in a file specified in the specification file. In the example, the first insertion of an action is for the symbol term 4 (position number 4) of production-rule 7 in production 38. In the actions, the actual structure carrying a symbol term is named SYMSTRUC. For example, SYMSTRUC(1) is the structure carrying the first symbol term in a production rule. A reference name to the name of the symbol term that the structure carries is SYMNM, for example SYMNM(2) is the name of the second symbol term of a production-rule. Note that actions are enclosed in curly brackets and there should be no blank lines in it.

INS_PRODRULE :

As the name of the keyword suggests, this keyword instructs the system to insert additional production-rules into an existing production. In the example, it refers to two insertions into production 67. The first inserted rule having two actions inserted, one for symbol term 1 and symbol term 3, while the other production-rule insertion included insertions of actions for symbol term 1 and symbol term 4. Any symbol term which the user does not wish to see printed (e.g. preprocessing keywords) in the output of the intended preprocessor should be post-fixed with (NP) in the inserted production-rule as is in the example.

ADD_NEWPROD :

This keyword is to be used to add a new production with a set of production-rules including actions. A complete production includes a L.H.S. non-terminal with production-rules on the R.H.S.. The specification format is similar to INS_PRODRULE@ as in the example but here no production number needs to be specified. An additional item to be specified is the non_terminal on the L.H.S..

SYNT_INCFILE :

This keyword instructs the system to include file/files that may contain variable declarations, routines and C codes that are used in the inserted actions in the productions.

RUNTIME_FILE :

This keyword facilitates the inclusion of runtime routines in the output of the preprocessor. It declares the name of the file that stores the routines. In the example, the file gcd_testmod.c contains test routines that provide the necessary pre and post condition checks.

TEXT_EXP :

This keyword provides the user the facility to include text or expand on the text in the input C program of the intended preprocessor. A number of text expansion variables are available. SYM refers to the current symbol which corresponds to the action which contains the required expansion. STRx e.g. STR1 is a string expansion variable and VALx is an integer expansion variable. The instruction is to expand a certain structure of the input C program as specified in the FN_HEAD section when instruction to do so is given in an action of a symbol term. TEXT_EXP is in fact a declaration of a text expansion routine that specifies a template in FN_HEAD with a template of the extended text in EX_TEXT. Note that the rest of the character strings other than SYM, STRx and VALx in EX_TEXT section will be printed as specified in the specification. The convention used in between double quotes in a printf statement is also required here i.e for example a newline (\n) in the EX_TEXT section will be written as \\n.

To illustrate the use of TEXT_EXP, a sample input program **gcd_mod.c** and the output program **gcd_modout.c** of the pre and post conditions preprocessor is included in the last section of this manual.

The system variables used in the actual implementation of the preprocessor are prefixed with the string "pp". The user is reminded of this to avoid name clashes occurring. Note also that a keyword specification is separated from another by a blank line.

PREFIX :

This instruction allows the user to prefix input strings in a program. This facility is useful especially when avoiding name clashes between the variable and function names of the input program; and variable and function names of the runtime routines.

NON_TERMINAL :

This keyword declares non-terminals to the system which

are to be used on the L.H.S. of new productions and inserted production rules.

To give an example of how the specification language is used, a simple specification file **ppcond_spec** of a pre- and post-condition preprocessor is given below :-

```
CNOTATION@COND    /* specifying commnent notation */

KEYWORD          /* specifying the keyword FNCHECK
@FNCHECK(NP)        which is not to be printed out in the output */
@

INS_ACTION       /* inserting an action for a symbol-term 3 in
PROD@38          production-rule 7 of production 38          */
RULE@7
ACT@4
@{
if (checkflag==1)
        {temp1=SYMNM(1);
         temp2=SYMNM(2);
         tstrptr=pplinkup(SYMSTRUC(3));
         thisptr=anarr;
         sprintf(thisptr,"%s%s%s",temp1,temp2,tstrptr);
        }
}
@

INS_ACTION
PROD@54
RULE@2
ACT@1
@{      /* conditional call to an expansion routine */
if ((levcount==1) && (checkflag==1))
        EXPAND_PRE(SYMSTRUC(1),thisptr);
}
@
RULE@4
ACT@2
@{
if ((levcount==1) && (checkflag==1))
        EXPAND_PRE(SYMSTRUC(2),thisptr);
}
@

INS_ACTION
PROD@55
RULE@1
```

```
ACT@1
@{
++levcount;
}
@

INS_ACTION
PROD@56
RULE@1
ACT@1
@{
--levcount;
}
@

INS_ACTION
PROD@62
RULE@5
ACT@3
@{
if (checkflag==1)
        EXPAND_POST(SYMSTRUC(1),SYMNM(2),thisptr);
}
@

INS_PRODRULE    /* inserting a production rule into production 67 */
PROD@67
PRULE
@FNCHECK(NP) declarator function_body
@
ACT@1
@{
checkflag=1;
}
@
ACT@3
@{
checkflag=0;
}
@
PRULE       /* inserting another production-rule */
@FNCHECK(NP) declaration_specifiers declarator function_body
@
ACT@1
@{
checkflag=1;
}
@
ACT@4
@{
```

```
checkflag=0;
}
@

SYNT_INCFILE@ppcdef.h  /* specifying variables/routines file(s) to
                          support the required preprocessing */
RUNTIME_FILE@ppcheck.h /* runtime file linked for runtime checks */

TEXT_EXP            /* specifying templates of expansion routines */
FN_HEAD@EXPAND_PRE(SYM,STR1)
EX_TEXT@SYM \\n pre_STR1);
FN_HEAD@EXPAND_POST(SYM,STR1,STR2)
EX_TEXT@post_STR2,STR1);\\n SYM
```

The System Environment

In the process of creating the required preprocessor, the system creates auxiliary files to assist it in the generation of the preprocessor. Most of the files will be deleted on completion of preprocessor except two files with the extensions **.lf** and **.yf**. Both of these files remain so that the user can check on the underlying structures of the lexical and syntactical stages. However, it is a good idea to open up a new directory when intending to use the system to safeguard against accidental overwriting of files.

Execution of System

To build a preprocessor **prepro** using a specification file **specfile**, type the command

psystem specfile prepro .

and the resultant preprocessor can be executed with an input program **input_prog** with the command

prepro input_prog .

Ambiguities, Conflicts and Errors.

If the input grammar rules specified in the specification file are ambiguous and causes conflicts, some messages will be reported by the system. The system invokes two disambiguating rules by default :
i) in a shift/reduce conflict, the default is to do the shift;
ii) in a reduce/reduce conflict, the default is to reduce by the earlier grammar rule. For further information and to understand how the system handles ambiguities, conflicts and errors, the user is advised to

look under the sections "Parser Operation", "Ambiguities and Conflicts" and "Error Handling" in the Unix's Yacc documentation.

The User Interface

A user friendly interface is available to the user in specifying the preprocessor. To initiate a dialogue session type **intface**. The specification will be written into specification file **sfile**. **sfile** can then be used as input to the system as explained under the section "Execution of System". An example of a dialogue session with the interface is given in Appendix VII.

This manual serves as a simple and brief introduction to CPBS. For a more detailed description of the specification language PSL, the preprocessor building system CPBS and the structure of productions, the user is advised to read Chapter 3 and 4 of the thesis.

Sample Input/Output

```
/*PPCOND FNCHECK*/
int MOD(num1,num2)
int num1,num2;

{
        int x;

        x=num1-num1/num2*num2;
        return x;
}



/*PPCOND FNCHECK*/
int GCD(a,b)
int a,b;

{
        int c,d,temp;

        c=a;
        d=b;
        while (d != 0)
        {
                temp=MOD(c,d);
                c=d;
                d=temp;
        }
```

```
        return c;
}
```

file gcd_mod.c

```
#include "ppcheck.h"
int MOD ( num1 , num2 ) int num1 , num2 ;

{ int x ;
 pre_MOD(num1 , num2 ); x = num1 - num1 / num2 * num2 ;
post_MOD(num1 , num2 ,x);
 return  x ;

} int GCD ( a , b ) int a , b ;

{ int c , d , temp ;
 pre_GCD(a , b ); c = a ;
d = b ;
while ( d != 0 )
{ temp = MOD ( c , d ) ;
c = d ;
d = temp ;

} post_GCD(a , b ,c);
 return  c ;

}
```

file gcd_modout.c (output)

```
#include "testlib.c"
void post_MOD(num1,num2,r)
int num1,num2,r;

{
        if (divides(num2,num1-r))
                printf("post_MOD satisfied.\n");
        else
                printf("post_MOD not satisfied, error - %d divides
                                %d rem %d\n",num2,num1,r);
}
/*-----------------------------*/
void post_GCD(a,b,r)
int a,b,r;
{
        int min,n,flag=0;
```

```
        if (divides(r,a) && divides(r,b))
        {
                if (r !=a && r != b)
                {
                        if (a<b)  min=a;
                        else min=b;
                        n=2;
                        while (n*r<min)
                        {
                                if (divides(n*r,a) && divides(n*r,b))
                                        flag=1;
                                else ++n;
                        }
                }
        }
        else flag=1;
        if (flag == 0)  printf("post_GCD satisfied.\n");
        else
           printf("post_GCD not satisfied, gcd %d %d is not %d\n",a,b,r);
}


void pre_MOD()

{
}

void pre_GCD()

{
}
```

file ppcheck.h

APPENDIX VI

Sample Input/Output of Preprocessor Examples and an Interface Dialogue

File : Cmain.c - input to generic package preprocessor (given below)

```
/*PPGPKG

GETPKG(CSTACK)
FUNC(char,pop);
FUNC(int,push);
FUNC(bool,empty_stack);
FUNC(bool,full_stack);
char top_of_stack;
END_GETPKG(CSTACK) */

main()
{
 FILE *ifp=fopen("INFILE","r");
 FILE *ofp=fopen("OUTFILE","w");

 /*PPGPKG getCSTACK();*/
 while (((*CSTACK.push)(getc(ifp)))!=EOF)
  {
   if (CSTACK.top_of_stack=='*' || (*CSTACK.full_stack)())
     while (((*CSTACK.empty_stack)())==FALSE)
        fprintf(ofp,"%c",(*CSTACK.pop)());
  }
 fprintf(ofp,"\n");
}
```

File : Cmainout.c : preprocessed output of Cmain.c (given below)

```
#include "gen_pkg.h"

extern struct {

FUNC ( char , pop ) ;

FUNC ( int , push ) ;

FUNC ( bool , empty_stack ) ;

FUNC ( bool , full_stack ) ;
char top_of_stack ;

} CSTACK;
 main ( )
```

```
{ FILE * ifp = fopen ( "INFILE" , "r" ) ;
FILE * ofp = fopen ( "OUTFILE" , "w" ) ;
getCSTACK ( ) ;
while ( ( ( * CSTACK . push ) ( getc ( ifp ) ) ) != EOF )
{ if ( CSTACK . top_of_stack == '*' || ( * CSTACK . full_stack ) ( ) )
while ( ( ( * CSTACK . empty_stack ) ( ) ) == FALSE )
fprintf ( ofp , "%c" , ( * CSTACK . pop ) ( ) ) ;

} fprintf ( ofp , "\n" ) ;

}
```

File : inst_CSTACK.h - input to generic package preprocessor (given below)

```
PKGINST(STACK,CSTACK)
INST(STACK_SIZE,1000)
INST(STACK_TYPE,char)
PKGLINK(getCSTACK)
END_PKGINST(STACK)
```

File : inst_CSTACKout.h - preprocessed output (given below)

```
#include "gen_pkg.h"

#define STACK CSTACK
#define STACK_SIZE 1000
#define STACK_TYPE char
#define LINK getCSTACK
#include "genpkg_STACK"
```

File : Rmain.c - input to generic package preprocessor (given below)

```
#include "rec.h"
#include <string.h>

/*PPGPKG
GETPKG(RSTACK)
FUNC(struct rec,pop);
FUNC(int,push);
FUNC(bool,empty_stack);
FUNC(bool,full_stack);
struct rec top_of_stack;
END_GETPKG(RSTACK)*/


main()
{
```

```
FILE *ifp=fopen("INFILE","r");
FILE *ofp=fopen("OUTFILE","w");
struct rec arec1,arec2;

/*PPGPKG getRSTACK();*/
while (!(feof(ifp)))
 { fscanf(ifp,"%s %d",arec1.name,&arec1.age);
   (*RSTACK.push)(arec1);
  if (strcmp(RSTACK.top_of_stack.name,"*")==0  || (*RSTACK.full_stack)())
   {arec2=(*RSTACK.pop)();
    while (((*RSTACK.empty_stack)())==FALSE)
      {arec2=(*RSTACK.pop)();
       fprintf(ofp,"%-10s %2d\n",arec2.name,arec2.age);
      }
   }
  }
}
```

File : Rmainout.c - Preprocessed output of Rmain.c (given below)

```
#include "gen_pkg.h"
struct rec
{ char name [ 10 ] ;
int age ;

} ;
extern char * strcpy ( ) , * strncpy ( ) , * strcat ( ) , * strncat ( ) , * strchr ( ) ,
                                        * strrchr ( ) , * strpbrk ( ) , * strtok ( ) ;
extern int strcmp ( ) , strncmp ( ) , strlen ( ) , strspn ( ) , strcspn ( ) ;

extern struct {

FUNC ( struct rec , pop ) ;

FUNC ( int , push ) ;

FUNC ( bool , empty_stack ) ;

FUNC ( bool , full_stack ) ;
struct rec top_of_stack ;

} RSTACK;
 main ( )
{ FILE * ifp = fopen ( "INFILE" , "r" ) ;
FILE * ofp = fopen ( "OUTFILE" , "w" ) ;
struct rec arec1 , arec2 ;
getRSTACK ( ) ;
while ( ! ( feof ( ifp ) ) )
{ fscanf ( ifp , "%s %d" , arec1 . name , & arec1 . age ) ;
( * RSTACK . push ) ( arec1 ) ;
if ( strcmp ( RSTACK . top_of_stack . name , "*" ) == 0 || ( * RSTACK . full_stack ) ( ) )
{ arec2 = ( * RSTACK . pop ) ( ) ;
```

```
while ( ( ( * RSTACK . empty_stack ) ( ) ) == FALSE )
{ arec2 = ( * RSTACK . pop ) ( ) ;
fprintf ( ofp , "%-10s %2d\n" , arec2 . name , arec2 . age ) ;

}
}
}
}
```

File : inst_RSTACK.h - input to generic package preprocessor (given below)

```
#include "rec.h"
PKGINST(STACK,RSTACK)
INST(STACK_SIZE,100)
INST(STACK_TYPE,REC)
PKGLINK(getRSTACK)
END_PKGINST(STACK)
```

File : inst_RSTACKout.h - preprocessed output (given below)

```
#include "gen_pkg.h"
struct rec
{ char name [ 10 ] ;
int age ;

} ;

#define STACK RSTACK
#define STACK_SIZE 100
#define STACK_TYPE struct rec
#define LINK getRSTACK
#include "genpkg_STACK"
```

ft B
Set of Simple Input Programs to Index-Checker, the input followed by the output

INPUT : A

```
/* AN ORDINARY C COMMENT */
main()

{int a[4][5],/*PPSUF SINGL_ARR */ b1[20],b2[200];

b1[10]=150;

/*PPSUF START_CHECK */
```

```
b1[10]=200;

/*PP A PREPROCESSING COMMENT */

/* A COMMENT LINE*/

b2[b1[10]]=500;

/*PPSUF END_CHECK */

printf("\nThe value is %d\n",b2[b1[10]]);

}
```

OUTPUT : A

```
#include "suf.rtm"
main ( )
{ int a [ 4 ] [ 5 ] , b1 [ 20 ] , b2 [ 200 ] ;
b1 [ 10 ] = 150 ;
 b1 [ (sfx[1]=10)<0 ||
          sfx[1]>=20?sfxxerror(sfx[1],9,"b1"):sfx[1] ] = 200 ;
/*PP A PREPROCESSING COMMENT */

b2 [ b1 [ (sfx[1]=10)<0 ||
          sfx[1]>=20?sfxxerror(sfx[1],15,"b1"):sfx[1] ] ] = 500 ;
 printf ( "\nThe value is %d\n" , b2 [ b1 [ 10 ] ] ) ;

}
```

INPUT : B

```
/*PP DEMONSTRATES PREPROCESSING WITH COM_CHECK */

/*PPSUF COM_CHECK*/
main()

{
int a[4][5],b1[20],b2[200];

b1[10]=150;


b2[b1[10]]=500;


printf("\nThe value is %d\n",b2[b1[10]]);

}
```

```
/* END OF DEMONSTRATION PROGRAM */
```

OUTPUT : B

```
#include "suf.rtm"
/*PP DEMONSTRATES PREPROCESSING WITH COM_CHECK */

 main ( )
{ int a [ 4 ] [ 5 ] , b1 [ 20 ] , b2 [ 200 ] ;
b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],9,"b1"):sfx[1] ] = 150 ;
b2 [ (sfx[2]=b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],12,"b1"):sfx[1] ])< 0 ||
        sfx[2]>=200?sfxxerror(sfx[2],12,"b2"):sfx[2] ] = 500 ;
printf ( "\nThe value is %d\n" , b2 [ (sfx[2]=b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],15,"b1"):sfx[1] ])< 0 ||
        sfx[2]>=200?sfxxerror(sfx[2],15,"b2"):sfx[2] ] ) ;

}
```

INPUT : C

```
/*PP A DEMONSTRATION FOR SCOPE_CHECK INSTRUCTION AND PREFIXING*/

main()
{ int b1[20],b2[200],sfx[100];

b1[10]=150;

{/*PPSUF SCOPE_CHECK */
int a[5][5];

a[2][3]=b1[10];

}

b2[b1[10]]=500;

/* THIS COMMENT WILL DISAPPEAR */

printf("\nThe value is %d\n",b2[b1[10]]);

}
```

OUTPUT : C

```
#include "suf.rtm"
/*PP A DEMONSTRATION FOR SCOPE_CHECK INSTRUCTION AND PREFIXING*/

main ( )
{ int b1 [ 20 ] , b2 [ 200 ] , sfxsfx [ 100 ] ;
b1 [ 10 ] = 150 ;

{ int a [ 5 ] [ 5 ] ;
a [ (sfx[1]=2)<0 ||
        sfx[1]>=5?sfxxerror(sfx[1],11,"a"):sfx[1] ] [ (sfx[2]=3)<0 ||
        sfx[2]>=5?sfxxerror(sfx[2],11,"a"):sfx[2] ] = b1 [ 10 ] ;

} b2 [ b1 [ 10 ] ] = 500 ;
printf ( "\nThe value is %d\n" , b2 [ b1 [ 10 ] ] ) ;

}
```

INPUT : D

```
/*PP DEMONSTRATION OF SINGL_ARR INSTRUCTION */

main()

{int /*PPSUF SINGL_ARR */ a[4],/*PP SUF SINGL_ARR - INVALID*/ b[20];


/*PPSUF START_CHECK */
a[10]=200;

a[b[10]]=500;

/*PPSUF END_CHECK */

printf("\nThe value is %d\n",a[10]);

}
```

OUTPUT : D

```
#include "suf.rtm"
/*PP DEMONSTRATION OF SINGL_ARR INSTRUCTION */

main ( )
{ int a [ 4 ] , /*PP SUF SINGL_ARR - INVALID*/

b [ 20 ] ;
 a [ (sfx[1]=10)<0 ||
        sfx[1]>=4?sfxxerror(sfx[1],9,"a"):sfx[1] ] = 200 ;
a [ (sfx[1]=b [ 10 ])<0 ||
```

```
        sfx[1]>=4?sfxxerror(sfx[1],11,"a"):sfx[1] ] = 500 ;
 printf ( "\nThe value is %d\n" , a [ 10 ] ) ;

}
```

INPUT : E

```
/*PP A DEMONSTRATION OF THE
  ALL_ARR INSTRUCTION      */

main()

{/*PPSUF ALL_ARR */ int a[4][5],b1[20],b2[200];

b1[10]=150;

/*PPSUF START_CHECK */
b1[10]=200;

b2[b1[10]]=500;

/*PPSUF END_CHECK */

printf("\nThe value is %d\n",b2[b1[10]]);

}
```

OUTPUT : E

```
#include "suf.rtm"
/*PP A DEMONSTRATION OF THE

  ALL_ARR INSTRUCTION      */

main ( )
{ int a [ 4 ] [ 5 ] , b1 [ 20 ] , b2 [ 200 ] ;
b1 [ 10 ] = 150 ;
 b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],10,"b1"):sfx[1] ] = 200 ;
b2 [ (sfx[2]=b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],12,"b1"):sfx[1] ])< 0 ||
        sfx[2]>=200?sfxxerror(sfx[2],12,"b2"):sfx[2] ] = 500 ;
 printf ( "\nThe value is %d\n" , b2 [ b1 [ 10 ] ] ) ;

}
```

INPUT : F

```
/*ONLY CHECKS ON A SINGLE ARRAY :
  THIS COMMENT WILL NOT BE PRINTED
  TO THE OUTPUT                 */

main()

{int a[4][5],/*PPSUF CHECK_ON*/b1[20],b2[200];

b1[10]=150;

b1[10]=200;

b2[b1[10]]=500;


printf("\nThe value is %d\n",b2[b1[10]]);

}


OUTPUT : F


#include "suf.rtm"
main ( )
{ int a [ 4 ] [ 5 ] , b1 [ 20 ] , b2 [ 200 ] ;
b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],7,"b1"):sfx[1] ] = 150 ;
b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],9,"b1"):sfx[1] ] = 200 ;
b2 [ b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],11,"b1"):sfx[1] ] ] = 500 ;
printf ( "\nThe value is %d\n" , b2 [ b1 [ (sfx[1]=10)<0 ||
        sfx[1]>=20?sfxxerror(sfx[1],14,"b1"):sfx[1] ] ] ) ;

}
```

file suf.rtm :

```
static int sfx[100];

static int sfxxerror(num,lc,string)
int num,lc;
char *string;
{printf("** ERROR on line no %d:index %d exceeds array size of %s.\n",lc,num,string);
return(num);
}
```

Demonstration of Piping Preprocessors genpkg and indchecker

INPUT : pipeeg.c

```
/*PP PIPING DEMONSTRATION*/

#include "rec.h"
#include <string.h>

/*PPGPKG
GETPKG(RSTACK)
FUNC(struct rec,pop);
FUNC(int,push);
FUNC(bool,empty_stack);
FUNC(bool,full_stack);
struct rec top_of_stack;
END_GETPKG(RSTACK)*/


main()
{
 FILE *ifp=fopen("INFILE","r");
 FILE *ofp=fopen("OUTFILE","w");
 struct rec arec1,arec2;

 /*PPGPKG getRSTACK();*/


{ int b1[20],b2[200],sfx[100];

b1[10]=150;

{/*PPSUF SCOPE_CHECK */
int a[5][5];

a[2][3]=b1[10];

}

b2[b1[10]]=500;


/* THIS COMMENT WILL DISAPPEAR */


printf("\nThe value is %d\n",b2[b1[10]]);

}

 while (!(feof(ifp)))
 { fscanf(ifp,"%s %d",arec1.name,&arec1.age);
   (*RSTACK.push)(arec1);
  if (strcmp(RSTACK.top_of_stack.name,"*")==0 || (*RSTACK.full_stack)())
   {arec2=(*RSTACK.pop)();
```

```
    while (((*RSTACK.empty_stack)())==FALSE)
      {arec2=(*RSTACK.pop)();
       fprintf(ofp,"%-10s %2d\n",arec2.name,arec2.age);
      }
   }
  }
}
```

OUTPUT : genpkg pipeeg.c | indchecker

```
#include "suf.rtm"
typedef struct
{ unsigned char * _ptr ;
int _cnt ;
unsigned char * _base ;
char _flag ;
char _file ;

} FILE ;
extern FILE _iob [ 20 ] ;
extern FILE * fopen ( ) , * fdopen ( ) , * freopen ( ) , * popen ( ) , * tmpfile ( ) ;
extern long ftell ( ) ;
extern void rewind ( ) , setbuf ( ) ;
extern char * ctermid ( ) , * cuserid ( ) , * fgets ( ) , * gets ( ) , * tempnam ( ) , * tmpnam ( ) ;
extern unsigned char * _bufendtab [ ] ;

/*PP PIPING DEMONSTRATION*/

struct rec
{ char name [ 10 ] ;
int age ;

} ;
extern char * strcpy ( ) , * strncpy ( ) , * strcat ( ) , * strncat ( ) , * strchr ( ) ,
                                    * strrchr ( ) , * strpbrk ( ) , * strtok ( ) ;
extern int strcmp ( ) , strncmp ( ) , strlen ( ) , strspn ( ) , strcspn ( ) ;
extern struct
{ struct rec ( * pop ) ( ) ;
int ( * push ) ( ) ;
int ( * empty_stack ) ( ) ;
int ( * full_stack ) ( ) ;
struct rec top_of_stack ;

} RSTACK ;
main ( )
{ FILE * ifp = fopen ( "INFILE" , "r" ) ;
FILE * ofp = fopen ( "OUTFILE" , "w" ) ;
struct rec arec1 , arec2 ;
getRSTACK ( ) ;

{ int b1 [ 20 ] , b2 [ 200 ] , sfxsfx [ 100 ] ;
```

```
b1 [ 10 ] = 150 ;

{ int a [ 5 ] [ 5 ] ;
a [ (sfx[1]=2)<0 ||
        sfx[1]>=5?sfxxerror(sfx[1],137,"a"):sfx[1] ] [ (sfx[2]=3)<0 ||
        sfx[2]>=5?sfxxerror(sfx[2],137,"a"):sfx[2] ] = b1 [ 10 ] ;

} b2 [ b1 [ 10 ] ] = 500 ;
printf ( "\nThe value is %d\n" , b2 [ b1 [ 10 ] ] ) ;

} while ( ! ( ( ( ifp ) -> _flag & 0020 ) ) )
{ fscanf ( ifp , "%s %d" , arec1 . name , & arec1 . age ) ;
( * RSTACK . push ) ( arec1 ) ;
if ( strcmp ( RSTACK . top_of_stack . name , "*" ) == 0 || ( * RSTACK . full_stack ) ( ) )
{ arec2 = ( * RSTACK . pop ) ( ) ;
while ( ( ( * RSTACK . empty_stack ) ( ) ) == 0 )
{ arec2 = ( * RSTACK . pop ) ( ) ;
fprintf ( ofp , "%-10s %2d\n" , arec2 . name , arec2 . age ) ;

}
}
}
}
```

OUTPUT : indchecker pipeeg.c | genpkg

```
#include "gen_pkg.h"
static int sfx [ 100 ] ;
static int sfxxerror ( num , lc , string ) int num , lc ;
char * string ;

{ printf ( "** ERROR on line no %d:index %d exceeds array size of %s.\n" , lc , num , string ) ;
return ( num ) ;

} /*PP PIPING DEMONSTRATION*/

struct rec
{ char name [ 10 ] ;
int age ;

} ;
extern char * strcpy ( ) , * strncpy ( ) , * strcat ( ) , * strncat ( ) , * strchr ( ) ,
                                * strrchr ( ) , * strpbrk ( ) , * strtok ( ) ;
extern int strcmp ( ) , strncmp ( ) , strlen ( ) , strspn ( ) , strcspn ( ) ;

extern struct {

FUNC ( struct rec , pop ) ;

FUNC ( int , push ) ;
```

```
FUNC ( bool , empty_stack ) ;

FUNC ( bool , full_stack ) ;
struct rec top_of_stack ;

} RSTACK;
 main ( )
{ FILE * ifp = fopen ( "INFILE" , "r" ) ;
FILE * ofp = fopen ( "OUTFILE" , "w" ) ;
struct rec arec1 , arec2 ;
getRSTACK ( ) ;

{ int b1 [ 20 ] , b2 [ 200 ] , sfxsfx [ 100 ] ;
b1 [ 10 ] = 150 ;

{ int a [ 5 ] [ 5 ] ;
a [ ( sfx [ 1 ] = 2 ) < 0 || sfx [ 1 ] >= 5 ? sfxxerror ( sfx [ 1 ] , 47 , "a" ) :
  sfx [ 1 ] ] [ ( sfx [ 2 ] = 3 ) < 0 || sfx [ 2 ] >= 5 ?
     sfxxerror ( sfx [ 2 ] , 47 , "a" ) : sfx [ 2 ] ] = b1 [ 10 ] ;

} b2 [ b1 [ 10 ] ] = 500 ;
printf ( "\nThe value is %d\n" , b2 [ b1 [ 10 ] ] ) ;

} while ( ! ( feof ( ifp ) ) )
{ fscanf ( ifp , "%s %d" , arec1 . name , & arec1 . age ) ;
( * RSTACK . push ) ( arec1 ) ;
if ( strcmp ( RSTACK . top_of_stack . name , "*" ) == 0 || ( * RSTACK . full_stack ) ( ) )
{ arec2 = ( * RSTACK . pop ) ( ) ;
while ( ( ( * RSTACK . empty_stack ) ( ) ) == FALSE )
{ arec2 = ( * RSTACK . pop ) ( ) ;
fprintf ( ofp , "%-10s %2d\n" , arec2 . name , arec2 . age ) ;

}
}
}
}
```

APPENDIX VII

A Dialogue with the Interface

```
$ intface
L#1 - Comment Notation (Y/N) :Y
L#1 - Comment Notation
      (1-4 alpha. characters)     :RP
L#1 - New Keyword(s) (Y/N)        :Y

L#2 - Keyword
      (a - quit )                 :REPEAT
L#2 - Keyword
      (a - quit )                 :UNTIL
L#2 - Keyword
      (a - quit )                 :a
L#1 - Prefixing (Y/N)             :N
L#1 - New Non-terminal(s) (Y/N)   :Y

L#2 - Non-terminal
      (a - quit )                 :repuntil
L#2 - Non-terminal
      (a - quit )                 :a

INSERT ACTION(S)/PRODUCTION RULE(S)/PRODUCTION SET(S)

-------------------------------------------
    a   -  insert Action(s)
    p   -  insert Production rule(s)
    s   -  insert new production Set(s)
    n   -  next production
    q      quit
-------------------------------------------

 Input Option :p
L#1  Production No. (0-quit)      :52

L#2  Type in Production Step
     ( a-quit)                    :repuntil

L#3  Action No.
     (e.g. 2 for Sym2, 0-quit)    :0
L#2  Type in Production Step
     ( a quit)                    :a

-------------------------------------------
    a   -  insert Action(s)
    p   -  insert Production rule(s)
    s   -  insert new production Set(s)
    n      next production
    q      quit
-------------------------------------------
```

```
L#2 - L.H.S.                          :rruntil
L#3 - Type In Production Step
      (q-quit)                        :REPEAT statement UNTIL LP expr RP'$C
L#4 -  Action No.
      (eg. 2 for Symb2, 0-quit)   :1
L#4 - Action
 (Enclose in curl brackets )
(Press RETURN to terminate input):{
TS_EX1(SYMSTRUC(1));
}
L#4 -  Action No.
      (eg. 2 for Symb2, 0-quit)   :3
L#4 - Action
 (Enclose in curl brackets )
(Press RETURN to terminate input):{
TS_EX2(SYMSTRUC(3));
}

L#4 -  Action No.
      (eg. 2 for Symb2, 0-quit)   :5
L#4 - Action
 (Enclose in curl brackets )
(Press RETURN to terminate input):{
TS_EX3(SYMSTRUC(5));
}

L#4 -  Action No.
      (eg. 2 for Symb2, 0-quit)   :0
L#3 - Type In Production Step
      (q-quit)                        :q

 ------------------------------------
    a   -  insert Action(s)
    p   -  insert Production rule(s)
    s   -  insert new Production Set(s)
    n   -  next Production
    q   -  quit
 ------------------------------------

 Input Option  :q
L#1 - Syntax Include File(s) (Y/N) :Y

L#2 - Filename
       (seperate with commas)     :var.h
L#1 - Runtime Variable(s)/Function(s) (Y/N)   :N
L#1 - Expand Text (Y/N)               :Y

L#2 - Function Head
      (q-quit)                        :TS_EX1(SYM)
L#2 - Expanded Text                   :do
L#2 - Function Head
      (q-quit)                        :TS_EX2(SYM)
L#2 - Expanded Text                   :while
L#2 - Function Head
      (q-quit)                        :TS_EX3(SYM)
L#2 - Expanded Text                   :!(SYM)
L#2 - Function Head
      (q-quit)                        :q
```

REFERENCES

[AHO86]
A.V. Aho, R. Sethi, J.D. Ullman, "Compilers : Principles, Techniques and Tools", Addison-Wesley Publication Company, 1986.

[AHO88]
A.V. Aho, B.W. Kernighan, P.J. Weinberger, "The AWK Programming Language", Addison-Wesley Publishing Company, 1988.

[BEIC84]
F.W. Beicheter, O. Herzog, H. Petzsch, "SLAN-4 : A Software Specification Design Language", IEEE Software Engineering, 1984.

[BOOC86]
G. Booch, "Software Engineering in Ada", 1986.

[BOUR83]
S.R. Bourne, "The Unix System", Prentice Hall International, 1983.

[BOYD83]
S. Boyd, "Modular C", SIGPLAN Notices, 18(4), 1983.

[BRON85]
C. Bron, T.J. Rossingh, "A Note on the Checking of Interfaces between Seperately Compiled Modules", SIGPLAN Notices, 20(8), 1985.

[BROW72]
P.J. Brown, "Extending High Level Languages by Macros - a practical evaluation", Software, 1972.

[BROW74]
P.J. Brown, "Macro processors and techniques for portable software", John Wiley & Sons Ltd., 1974.

[BROW79]
P.J. Brown, "Macro without tears", Software - Practice and Experience 9(6), 1979.

[CAMP73]
M. Campbell-Kelly, "An Introduction to Macros", Macdonald & Co.(Publishers) Ltd., 1974.

[CAMP78]
W.R. Campbell, "A compiler definition facility based on the syntactic macro", Computer Journal 21(1), 1978.

[CHEA66]
T.E. Cheatham, "The introduction of definitional facilities into higher level programming language, Proc. FJCC, AFIPS 29, 1966.

[CHEA69]
T.E. Cheatham, "Motivations for Extensible Languages", SIGPLAN Notices 4(8), 1969.

[CLAY82]
B.G. Claybrook, "A Specification Method for Specifying Data and Procedural Abstractions", IEEE Software Engineering, 1982.

[COLE81]
A.J. Cole, "Macro Processors - 2nd. Edition", Cambridge University Press, 1981.

[COX86]
B.J. Cox, "Object Oriented Programming : An Evolutionary Approach", Addison Wesley Publishing Company, 1986.

[DUTT85]
K. Dutta, "Modular programming in C : an approach and an example", SIGPLAN Notices 20(3), 1985.

[GEHA84]
N. Gehani, "Ada : An Advanced Introduction Including Reference Manual", Prentice Hall, 1984.

[GREE79]
S.R. Greenwood, "MACRO : A programming language", SIGPLAN Notices 14(12), 1979.

[GRUN86]
D. Grune, "Generic Packages in C", SIGPLAN Notices 21(8), 1986.

[HAYE86]
I.J. Hayes, "Specification Directed Module Testing", IEEE Software Engineering, 12(1), 1986.

[HIGL76]
C.J. Higley, "Type Checking in a Typeless Language", The Computer Journal 19(2), 1976.

[JACK83]
M. Jackson, "System Development", Prentice Hall International, 1983.

[JONE86]
C.B. Jones, "Systematic Software Development Using VDM", Prentice Hall International, 1986.

[KERN78]
B.W. Kernighan and D.M. Ritchie, "The C Programming Language", Prentice Hall, 1978.

[KIEB78]
R.B. Kieburtz, W. Barabash, C.R. Hill, "A Type-checking Program Linkage System for Pascal", Proceeding of the 3rd. International Conference on Software Engineering, 1978.

[LAYZ85]
P.J. Layzell, "The History of Macro Processors in Programming Language Extensibility", Computer Journal, 28(1), 1985.

[LEAV66]
B.M. Leavenworth, "Syntax Macros and Extended Translations", Comm. ACM 9(11),1966.

[LEBL79]
R.J. LeBlanc, C.N. Fisher, "On Implementing Seperate Compilation in Block-Structured Languages", ACM, Vol. 14, 1979.

[LISK75]
B.H. Liskov, S. Zilles, "Specification Techniques for Data Abstractions", IEEE Software Engineering, SE-1(1), 1975.

[MERTZ79]
J.R. Metzner, "A graded bibliography on macro systems and extensible languages", SIGPLAN Notices 14(1), 1979.

[NOLA70]
M. Nolan, "MPL : Macro Pre-Compiling Language", Software Age, April, 1970.

[PARN72]
D.L. Parnas, "A Technique for Software Module Specification with Examples", CACM, 15(5), 1972.

[PARN72b]
D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", CACM, 15(5), 1972.

[STRO82]
B. Stroustrup, "Classes : An Abstract Data Type Facility for C Language", SIGPLAN Notices, 17(1), 1982.

[STRO83]
B. Stroustrup, "Adding Classes to the C Language : An Exercise in Language Evolution", Software - Practice and Experience, Vol. 13, 1983.

[TRIA85]
J.M. Triance, P.J. Layzell, "Macro Processors for Enhancing High-Level Languages - Some Design Principles", Computer Journal 28(1), 1985.

[WEB86]
H. Weber, H. Ehrig, "Specification of Module Systems", IEEE Software Engineering, 12(7), 1986.

Unix System V - Release 2.0 Support Tools Guide, April, 1984. AT&T Bell Laboratories.